开源项目中文文档

Table of Contents

Java 开源项目中文文档

Spring 中文文档 3.1

Ver 1.6.1（build on 20111007）

南磊 译

电子邮件：nanlei1987@gmail.com 微博：http://weibo.com /nanlei1 987

第一部分 Spring framework 概述

Spring framework 是一个轻量级的解决方案，在构建一站式企业级应用程序上有很大的 潜能。Spring 是模块化的，允许你使用仅需要的部分，而不需要引入其余部分。你可以使用 IoC 容器，和 Struts 一起使用，而且你也可以仅仅使用 Hibernate 整合代码或者是 JDBC 抽象 层。Spring framework 支持声明式的事务管理，通过 RMI 或 Web Service 远程访问业务逻辑 代码，并且提供多种持久化数据的选择。它提供饿了一个全功能的 MVC 框架，允许你显式 地整合 AOP 到软件中。

Spring 被设计成非侵入式的，也就是说你的业务逻辑代码通常是不会对 Spring 框架本身 产生依赖的。在你的整合层面（比如数据访问层），一些依赖于数据访问技术和 Spring 的类 库是会存在的。但是，也很容易将这些依赖从你剩余的代码中分离出来。

本文档是 Spring 框架特性的参考指南。如果你有任何想法，建议或是对本文档的疑问， 请发送到用户邮件列表中或者是在线论坛中，论坛地址是 http://forum.springsource.org。

第 1 章 Spring Framework 介绍

Spring Framework 是一个 Java 平台，它提供了对用 Java 语言开发应用程序的一种广泛的 基础支持。Spring 本身来控制这个基础，那么你就可以集中精力于应用程序的开发了。

Spring 允许你从“普通 Java 对象（POJO）”来构建应用程序，并且将非侵入地企业级服 务应用于 POJO 中。这种能力不仅适用于 Java SE 编程模型，也适用于全部或部分的 Java EE。

作为应用程序的开发人员，下面就是可以使用 Spring 平台所含优点的例子：

编写 Java 方法来执行数据库事务而不需要处理相关的事务 API。

编写本地的 Java 方法来访问远程程序而不需要处理远程访问 API。

编写本地的 Java 方法来执行管理操作而不需要处理 JMX 的 API。

编写本地的 Java 方法来处理消息操作而不需要处理 JMS 的 API。

1.1 依赖注入和控制反转

背景

“问题是，[它们]反向控制哪一方面？”，2004 年，Martin Fowler 在他个人站点提出了这个关于控制反转（IoC）的问题。Fowler 建议重命名这个原则，使得它更好地自我解 释，同时提出了依赖注入。

要 深 入 了 解 IoC 和 DI ， 可 以 参 考 Fowler 的 文 章 ， 地 址 是 ：

http://mart inf owler.com/artic les/injection.html

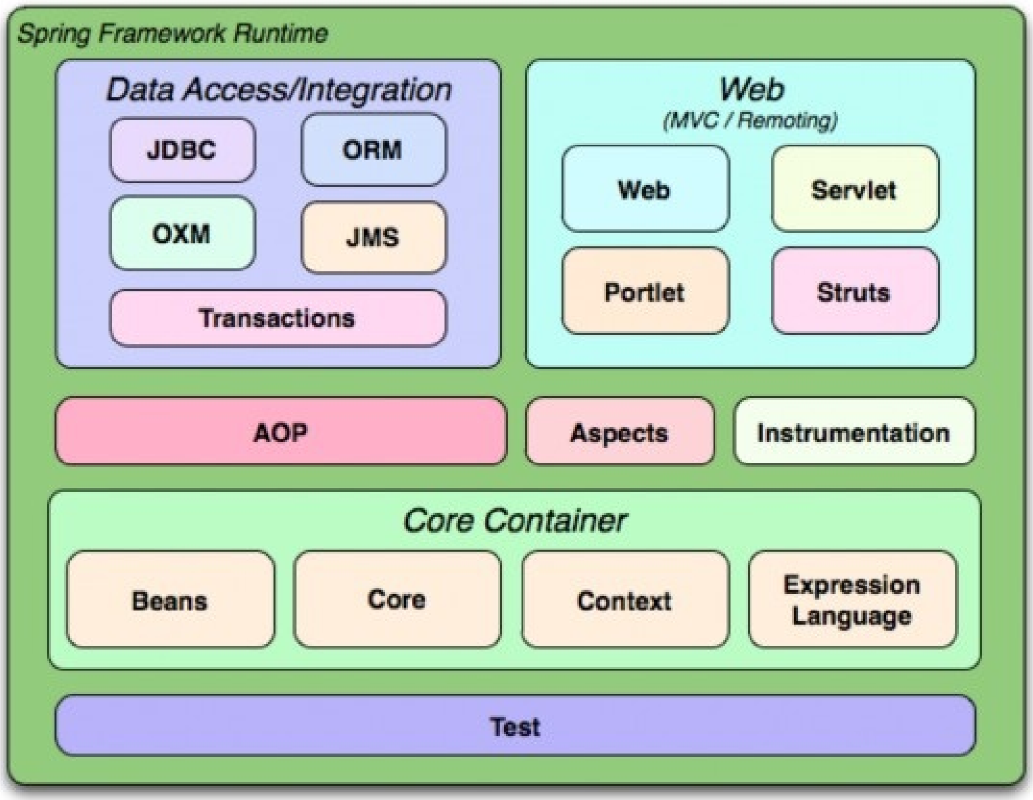
Java 应用程序 -- 一个宽松的术语，囊括了从被限制的 applet 到 n 层服务器端的企业级 应用程序的全部 – 典型的应用是，包含了组成独特应用程序的合作对象。那么在应用程序 中的这些对象就会有相互依赖关系。

尽管 Java 平台提供了丰富的应用程序开发功能，但是它也缺乏组织基本模块到整体的 方式，而是把这个任务留给了系统架构师和开发人员去解决。也就是说，你可以设计如工厂， 抽象工厂，构建者，装饰者和服务定位器等设计模式来组合各个类，以及构成该应用程序的 对象的实例。然而，这些模式都是最简单的：最佳的做法是给定一个名称，并且描述这个模 式做了些什么，在哪里可以应用它，它所强调的问题是什么等等。模式可以使得你必须自己 实现的最佳实践形式化。

Spring Framework 的控制反转（Inversion of Control，IoC）组件提供了组合不同的组件到 完整可用的应用程序的形式化方法。Spring Framework 编写了形式化的设计模式作为顶级对 象，你可以用来整合到你自己的应用程序中。很多组织和研究机构使用 Spring Framework 的 这个方式来设计健壮的，可维护的应用程序。

1.2 模块

Spring Framework 包含了很多特性，并且组织成 20 个模块。这些模块分为核心容器， 数据访问/整合，Web，AOP（Aspect Oriented Programming，面向切面编程），基础组件和测 试，在下图中来展示这些模块。



Spring 框架概要

1.2.1 核心容器

核心容器（4.1 节）包含了核心（Core），Bean 组件（Beans），上下文（Context）和表 达式语言（Expression Language）模块。

核心和 Bean（4.1 节）模块提供了框架部分的基础，包含 IoC 和依赖注入特性。

BeanFactory 是工厂模式的精密实现。它去掉了编程实现单例的需要，并允许你解除配置 信息，以及实际程序逻辑的特定依赖之间的耦合。

上下文（4.14 节）模块构建了由核心和 Bean（4.1 节）模块提供的坚实基础：它可以让 你以框架中的风格来访问对象，这和 JNDI 的注册是相似的。上下文模块集成了来自 Bean 模块的特性，并且加入了对国际化（比如使用资源束）的支持，事件传播，资源加载和 Servlet容器显式创建上下文。上下文模块也支持 Java EE 特性，比如 EJB，JMX 和基本的远程调用。

ApplicationContext 接口是上下文模块的焦点。

表达式语言（第 7 章）模块提供了强大的表达式语言，在运行时查询和操作对象图。这 是 JSP 2.1 规范中的统一表达式语言（Unified EL）的一个扩展。该表达式语言支持设置和获取属性值，属性定义，方法调用，访问数组，集合以及索引的上下文。支持逻辑和数字运算， 命名变量。还支持从 Spring 的 IoC 容器中以名称来获取对象。它也支持 list 的投影和选择操作，还有普通的 list 聚集。

1.2.2 数据访问/整合

数据访问/整合层由 JDBC，ORM，OXM，JMS 和事务模块组成。

JDBC（13.1 节）模块提供了 JDBC 抽象层，它移除了冗长的 JDBC 编码，但解析了数据库 提供商定义的特定错误代码。

ORM（14.1 节）模块提供了对流行的对象-实体映射 API 的整合层，包含 JPA（14.5 节），JDO（14.4 节），Hibernate（14.3 节）和 iBatis（14.6 节）。使用 ORM 包你就可以使用全部的O/R-映射框架并联合其它 Spring 提供的特性，比如之前所提到的简单声明式的事务管理特性。

OXM（第 15 章）模块提供了支持 JAXB，Castor，XMLBeans，JiBX，以及 Xstream 对对象/XML 映射实现的抽象层。

Java 消息服务（JMS，第 22 章）模块包含生成和处理消息的特性。

事务（第 11 章）模块支持对实现特定接口的类和所有 POJO（普通 Java 对象）的编程 式和声明式的事务管理。

1.2.3 Web

Web 层由 Web，Web-Servlet，Web-Struts 和 Web-Portlet 模块组成。

Spring 的 Web 模块提供了基本的面向 Web 整合的特性，比如文件上传功能，IoC 容器 的初始化使用了 Servlet 的监听器和变形的 Web 应用上下文。它还包含了和 Web 相关的 Spring远程调用支持的部分。

Web-Servlet 模块包含了 Spring 对 Web 应用的模型-视图-控制器（MVC，16.1 节）模式 的实现。Spring 的 MVC 框架提供了一个在领域模型代码和 Web 表单之间的整洁分离，并且 整合了其它所有 Spring 框架的特性。

Web-Struts 模块包含了使用 Spring 应用程序整合经典的 Strtus Web 层的支持类。要注意 这个支持从 Spring 3.0 开始就废弃了。那么可以考虑迁移应用程序到 Struts 2.0 和 Spring 的整 合或者是单纯的 Spring MVC 方案。

Web-Portlet 模块提供用于 portlet 环境和 Web-Servlet 模块功能镜像的 MVC 实现。

1.2.4 AOP 和基础组件

Spring 的 AOP（8.1 节）模块提供了 AOP 联盟-允许的的面向切面的编程实现，允许你定 义如方法-拦截器和横切点来整洁地解耦应该被分离的功能实现代码。使用源代码级的元数 据功能，也可以混合行为信息到代码中，这个方式和.NET 的属性很相似。

分离的 Aspects 模块提供对 AspectJ 的整合。

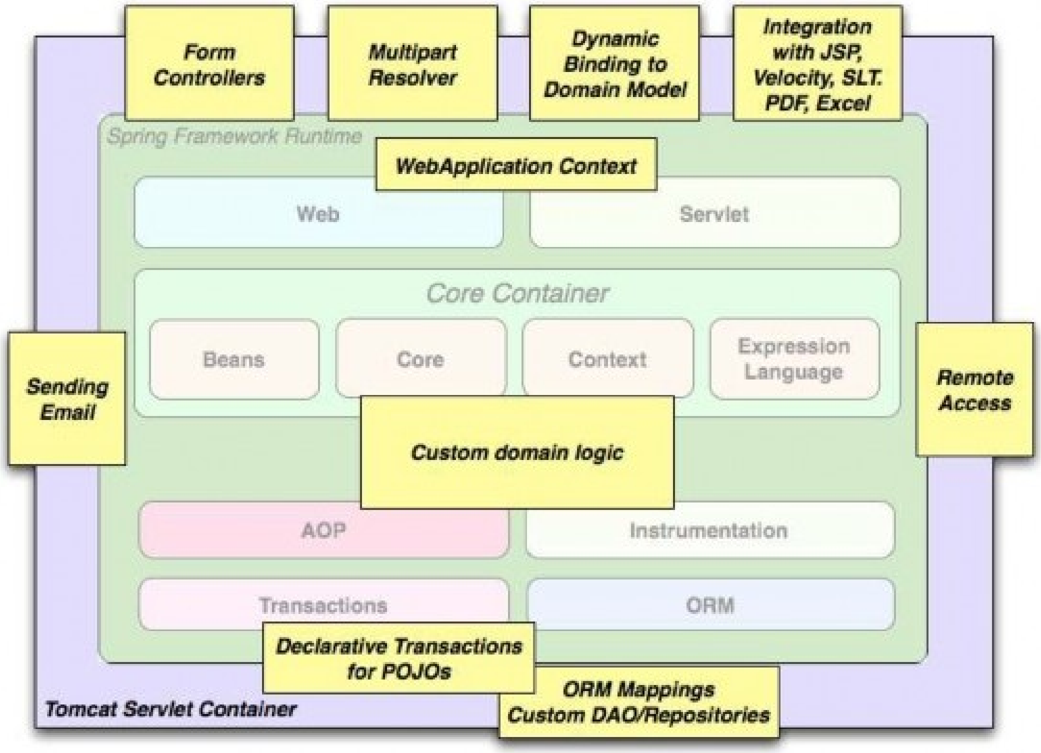
基础组件模块提供了对类的基础支持，还有类加载器的实现来用于特定的应用服务器。

1.2.5 测试

测试模块支持 Spring 组件，Junit 或 Test NG 的测试。它提供了 Spring 应用上下文的一致 加载并缓存这些上下文内容。它也提供 mock 对象，你可以用来孤立地测试代码。

1.3 使用方案

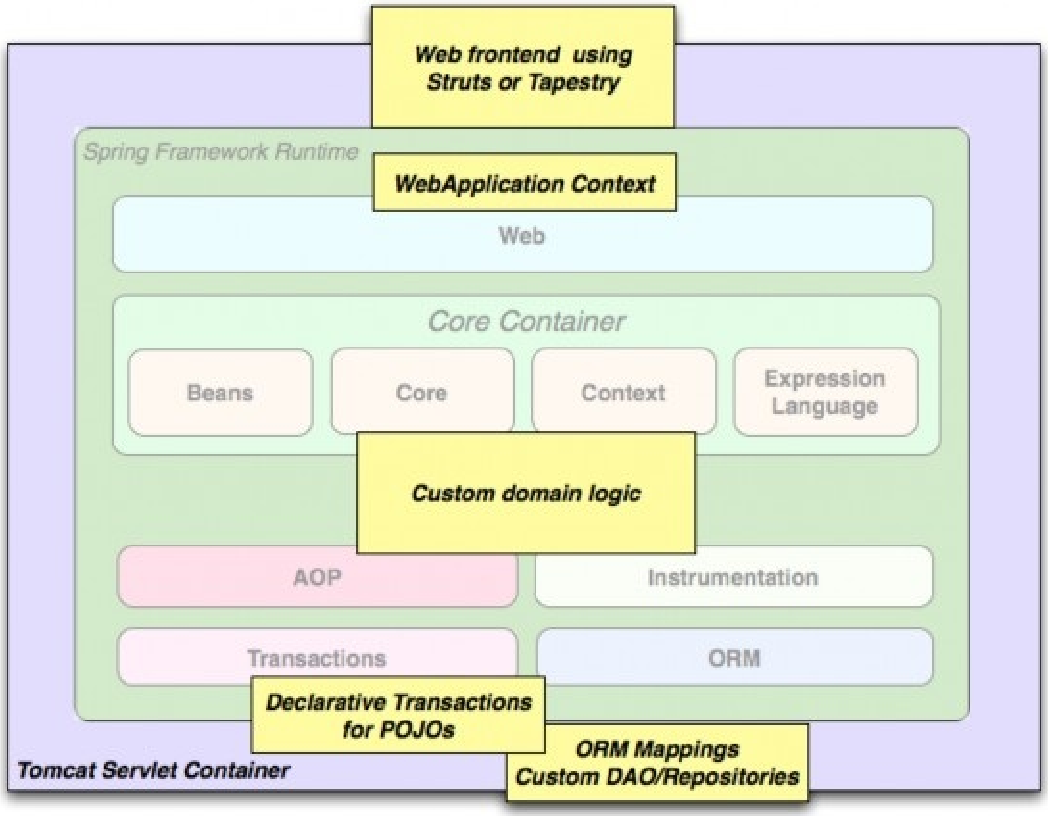
前面描述的模块使得 Spring 可以在很多方案中作为业务逻辑实现的选择，从 applet 到 使用了 Spring 的事务管理功能和 Web 框架整合，功能完善的企业级应用程序。



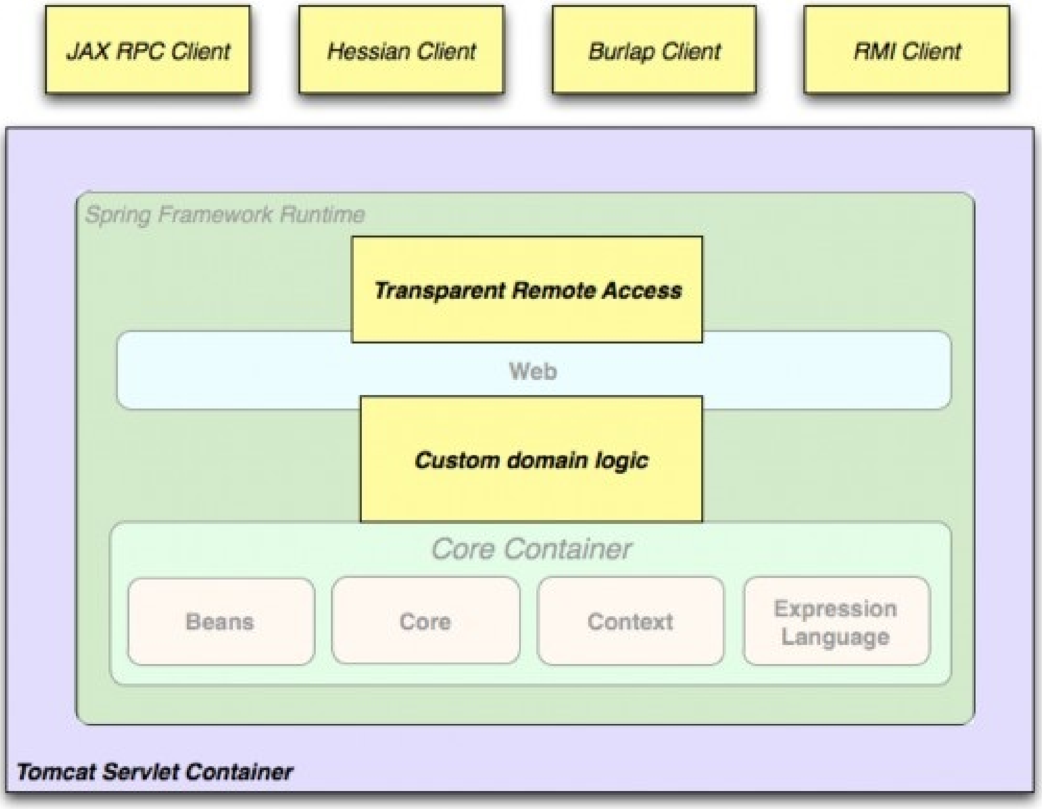
典型地功能完善的 Spring Web 应用

Spring 的声明式事务管理特性（11.5 节）使得 Web 应用程序可以全部事务化，就好像

使用了 EJB 容器管理的事务。全部的自定义业务逻辑可以使用简单的 POJO 来实现并且被 Spring 的 IoC 容器来管理。其它的服务包含对发送邮件的支持，验证对 Web 层独立，这可以 让你选择在哪里执行验证规则。Spring 的 ORM 支持对 JPA，Hibernate,JDO 和 iBatis 进行了整合；比如，当使用 Hibernate 时，你可以继续使用已有的映射文件和标准的 Hibernate 的 SessionFactory 配置 。表 单控 制器 无缝 地整 合了 Web 层 和领 域模 型， 也去 除了 ActionForm 或其它为领域模型转换 HTTP 参数的类的需要。

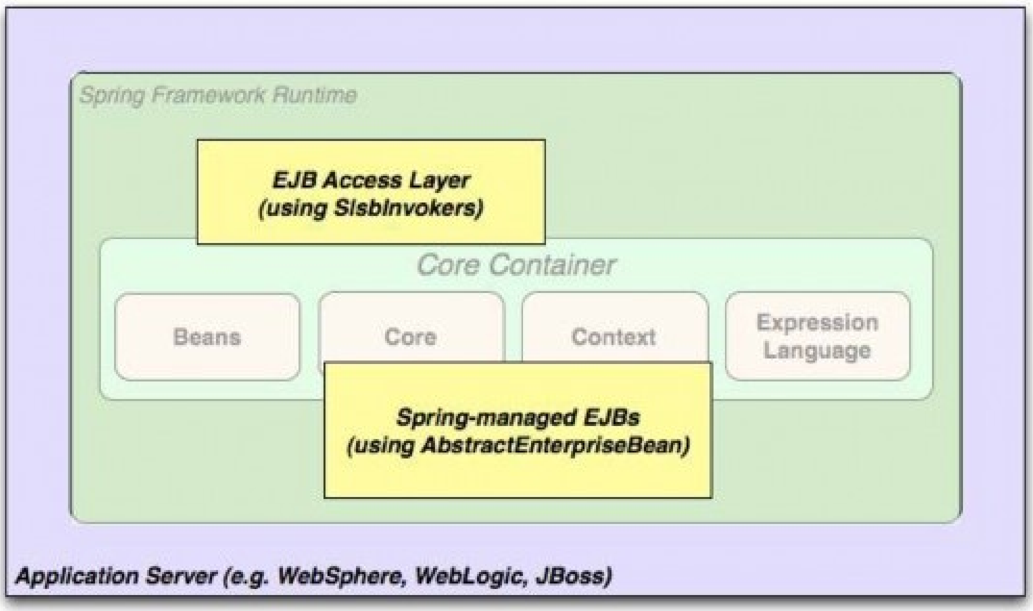


使用了第三方 Web 框架的 Spring 中间层 有时，开发和运行环境并不允许你完全转换到一个不同的框架中。Spring Framework 不强制你使用其中的部分；它并不是一个所有或没有的方案。已有使用了 WebWork，Struts， Tapestry 或其它 UI 框架构建的前端应用程序也可以使用基于 Spring 的中间层来进行整合， 这 就 允 许 你 去 使用 Spring 的 事 务 特 性 。 你 仅 仅 需 要 做 的 是 给 业 务 逻 辑 装 上 ApplicationContext，对整合的 Web 层使用 WebApplicationContext。



远程调用使用方案

当你需要通过 Web Service 访问已有的代码时，你可以使用 Spring 的 Hessian-，Burlap-，Rmi-或 JaxPpcProxyFactory 类。对已有的应用程序开启远程访问并不困难。



EJB – 包装已有的 POJO

Spring Framework 也提供了对企业级 Java Bean（EJB，译者注）的访问和抽象层（第 21 章），这就可以重用已有的 POJO，可以扩展、包装它们到无状态会话 bean，不安全的 Web 应用程序可能也需要声明式安全。

1.3.1 依赖管理和命名规约

依赖管理和依赖注入是不同的概念。要在应用程序中添加 Spring 的优雅特性（比如依 赖注入），你需要放置所需的类库（jar 文件），并且添加到运行时环境的类路径中，通常在 编译时也是需要这些类库文件的。这些依赖不是注入的虚拟组件，而是文件系统（通常来说 是这样）上的物理资源。依赖管理的过程包括定位资源，存储它们，并将它们添加到类路径 中。依赖可以是直接的（比如应用程序在运行时需要 Spring），或者是间接的（比如应用程 序需要的 commons-dbcp 组件还依赖于 commons-pool 组件）。间接的依赖也被认为是“过 度的”，而且那些依赖本身就难以识别和管理。

如果你决定使用 Spring，那么你需要获得 Spring 的 jar 包，其中要包括你所需要使用的 Spring 的模块。为了使用的便捷，Spring 被打包成各个模块的集合，尽可能地分离其中的相 互依赖，那么如果你不想编写 Web 应用程序，就不需要添加 spring-web 模块的 jar 包。要参 考 Spring 模 块 的 库 ， 本 指 南 使 用 了 一 个 可 以 速 记 的 命 名 规 约 ， spring- 或者 *spring*-.jar ，这里的“ * ” 代 表 了 各 模 块 的 短 名 称 （ 比 如 ， spring-core ， spring-webmvc，spring-jms 等）。实际上 jar 文件命名或许就是这种形式的（参考下 面的示例），但也有可能不是这种情况，通常它的文件名中还包含一个版本号（比如， spring-core-3.0.0.RELEASE.jar）。

通常，Spring 在四个不同的地方发布组件：

- 在社区下载点 http://www.springsource.org/dow nloads/comm unity。在这儿你可以找到所 有的 Spring jar 包，它们被压缩到一个 zip 文件中，可以自由下载。这里 jar 包的命名从 3.0 版本开始，都以 org.springframework.*-.jar 格式进行。

- Maven 的中央库，也是 Maven 默认检索的资源库，它并不会检索特殊的配置来使用。 很多 Spring 所依赖的常用类库也可以从 Maven 的中央库中获得，同时 Spring 社区的绝大多数用户都使用 Maven 作为依赖管理工具，这对于他们来说是很方便的。这里的 jar 包的命名规则是 spring-*-.jar 格式的，并且 Maven 里的 groupId 是 org.springframework。

- 企业级资源库（Enterprise Bundle Repository，EBR），这是由 SpringSource 组织运营的， 同时也提供了和 Spring 整合的所有类库。对于所有 Spring 的 jar 包及其依赖，这里也有 Maven 和 Ivy 的资源库，同时这里还有很多开发人员在使用 Spring 编写应用程序时能用到的其它大量常用类库。而且，发布的版本，里程碑版本和开发版本也都部署在这里。 这 里 jar 文 件 的 命 名 规 则 和 从 社 区 下 载 的 （ `org.springframework.*-<version>.jar` ）一致，并且所依赖的外部类库（不 是来自 SpringSource 的）也是使用的这种“长命名”的形式，并以 com.springsource 作为前缀。可以参考 FAQ 部分获取更多信息。

- 在 Amazon S3 为开发和里程碑版本发布（最终发布的版本这里也会有）而设置的公共Maven 资源库。Jar 文件的名称和 Maven 中央库是一样的，那么这里是获取 Spring 开发版本的地方，其它的类库是部署于 Maven 中央库的。 那么，首先你要决定的事情是如何管理这些依赖：很多人使用自动化的系统，比如 Maven 和 Ivy，但是你也可以手动下载所有的 jar 文件。当使用 Maven 或 Ivy 获取 Spring 时，之后你就需要决定从哪里来获取它们。通常来说，如果你关注 OSGi 的话，那么就使用 EBR，因为 它对所有的 Spring 依赖兼容

OSGi，比如 Hibernate 和 Freemarker。如果对 OSGi 不感兴趣，那么使用哪个下载都是可以的，但是他们也各有利弊。通常来讲，为你的项目选择一个或者备用的一个；但是请不要混用。因为相对于 Maven 中央库而言，EBR 组件非常需要一个不同的命名规则，那么这就特别重要了。

表 1.1 Maven 中央库和 SpringSource EBR 资源库的比较

特性	Maven 中央库	EBR
OSGi 的兼容	不明确	是
组件数量	成千上万；所有种类	几百个；是 Spring 整合中会用到的
一致的命名规约	没有	有
命名规约：GroupId	相异。新组件通常使用域名，比如 org.slf4j。老组件通常仅仅使用组件名，比 如 log4j。	原 始 的 域 名 或 主 包 的 根 路 径 ，比 如 org.springframework
命名规约：ArtifactId	相异。通常是项目或模块 名，使用连字符"-"分隔， 比如 spring-core，log4j。	捆绑符号名称，从主包的路径分离，比如 org.springframework.beans。如果 jar 需要修补以保证兼容 OSGi，那么就附加 com.springsource ， 比 如 com.springsource.org.apache.log4j
命名规约：Version	相异。很多新的组件使用 m.m.m 或 m.m.m.X（m 是数字，X 是文本）。老的使 用 m.m。有一些也不是。 顺序是确定的但通常并不 可靠，所以不是严格的可靠。	OSGi 版本数字 m.m.m.X，比如 3.0.0.RC3。 文本标识符使用相同的数字值规定版本 的字母顺序。
发布	通常是自动通过 rsync 或源码控制更新。项目作者 可以上传独立的 jar 文件到 JIRA。	手动（由 SpringSource 控制的 JIRA）
质量保证	根据政策。精确度是作者的责任。	宽泛的 OSGi 清单，Maven POM 和 Ivy 元数据。QA 由 Spring 团队执行。
主机	Contegix 提供。由 Sonatype 和一些镜像构成。	由 SpringSource 的 S3 构成
搜索工具	很多	http://www.springsource.com/repository
和 SpringSource 工具整合	通过和 Maven 依赖管理的 STS 来整合	通过 Maven ，Roo ，CloudFoundry 的 STS 进行宽泛整合

1.3.1.1 Spring 依赖和基于 Spring

尽管 Spring 提供对整合的支持，以及对大量企业级应用及其外部工具的支持，那么它 也有心保持它的强制依赖在一个绝对小的数目上：你不需要为了简单的使用去定位并下载

（甚至是自动地）大量的 jar 来使用 Spring。对于基本的依赖注入那只需要一个必须的外部 依赖，就是日志（可以参考下面关于日志的深入介绍）。

下面，我们来概述一下配置一个基于 Spring 的应用程序所需的基本配置，首先使用

Maven 和 Ivy。在所有的示例中，如果有哪一点不清楚，可以参考你所使用的依赖管理系统 的相关文档，或者参考一些示例代码 – Spring 本身在构建时使用了 Ivy 来管理依赖，而我们

大多数的示例却使用 Maven。

1.3.1.2 Maven 依赖管理

如果你正使用 Maven 来进行依赖管理，那么你就不需要明确地提供日志依赖。比如， 要为应用程序配置创建应用上下文并使用依赖注入特性，那么，你的 Maven 依赖配置文件 可以是这样的：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

就是这么简单。要注意的是，如果你在编译代码时不需要使用 Spring 的 API，那么 scope

元素可以声明为 runtime，这就可以用于典型的依赖注入用例。 在上面的示例中，我们使用了 Maven 中央库中的命名规约，那么它就可以在 Maven 中

央库或 SpringSource S3 的 Maven 资源库中起作用。要使用 S3 的 Maven 资源库（比如里程碑

版本或开发快照）版本，那就需要在 Maven 的配置文件中明确地指定资源库的位置。对于 完整发布版，配置如下：

```
<repositories>
    <repository>
        <id>com.springsource.repository.maven.release</id>
        <url>http://maven.springframework.org/release/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

对于里程碑版本配置如下：

```
<repositories>
    <repository>
        <id>com.springsource.repository.maven.milestone</id>
        <url>http://maven.springframework.org/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

对于开发快照版本配置如下：

```
<repositories>
    <repository>
        <id>com.springsource.repository.maven.snapshot</id>
        <url>http://maven.springframework.org/snapshot/</url>
        <snapshots><enabled>true</enabled>< /snapshots>
    </repository>
</repositories>
```

要使用 SrpingSource 的 EBR 的话，你还需要一个对依赖不同的命名规约。名称通常很容 易去猜测，比如这个示例中，就是：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>org.springframework.context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

你也可能需要明确地声明资源库的位置（仅仅 URL 是重要的）：

```
<repositories>
    <repository>
        <id>com.springsource.repository.bundles.release</id>
        <url>http://repository.springsource.com/maven/bundles/re
lease/</url>
    </repository>
</repositories>
```

如果你要手动地去管理依赖，在上面这个声明的资源库 URL 是不可以浏览的，但是它 也有一个用户界面，地址是 http://www.springsource.com/repositor y，这可以用来搜索和下 载依赖。它也有 Maven 和 Ivy 配置的代码片段，你可以复制下来并粘贴到你所使用的工具中， 很方便使用。

1.3.1.3 Ivy 依赖管理

如果你使用 Ivy 来管理依赖，那么有一些简单的命名规约和配置选项。

要配置 Ivy 定位到 SpringSource EBR 中，需要添加如下的解析器元素到你的配置文件

ivysettings.xml 中：

```
<resolvers>
    <url name="com.springsource.repository.bundles.release">
        <ivy pattern="http://repository.springsource.com/ivy/bun
dles/release
        /
        [organisation]/[module]/[revision]/[artifact] -[revision
].[ext]"
        />
        <artifact pattern="http://repository.springsource.com/iv
y/bundles/release
        /
        [organisation]/[module]/[revision]/ [artifact]-[revision
].[ext]"
        />
    </url>
    <url name="com.springsource.repository.bundles.external">
        <ivy pattern="http://repository.springsource.com/ivy/bun
dles/externa l/
        [organisation]/[module]/[revision]/[artifact]-[revision]
.[ext]"
        />
        <artifact pattern="http://repository.springsource.com/iv
y/bundles/externa l/
        [organisation]/[module]/[revision]/[artifact]-[revision]
.[ext]"
        />
    </url>
</resolvers>
```

上面的 XML 并不是合法的格式，因为行太长了 – 如果你要复制粘贴，那么要移除中部

url 模式部分结尾额外的行。（本文档中已经去除）

一旦 Ivy 配置好去查看 EBR，那么添加依赖就很简单了。只要拉起在资源库浏览器中的 捆绑详细信息页面，你就会发现为你准备好的 Ivy 代码片段，那么就可以将它包含到你的依

赖部分中了。比如（在 ivy.xml 中）：

```
<dependency org="org.springframework" name="org.springframework.
core" rev="3.0.0.RELEASE"
conf="compile->runtime"/>
```

1.3.2 日志

对于 Spring 来说，日志是一个非常重要的依赖，因为 a)这是唯一强制的外部依赖，b) 开发人员都会想看到他们所使用的工具的一些输出内容，而且 c)Spring 整合了多种实用工具， 它们都会选择一种日志依赖包。应用程序开发人员的目标之一

就是在核心位置对整个应用程 序有一个统一的日志配置，包括对所有的外部组件。因为日志框架有多种选择，那么这就可 能有些难以确定了。

Spring 中强制的日志依赖包是 Jakarta 的 Commons Logging API（JCL）。我们对 Spring 的 编译是基于 JCL 的，而且我们使扩展了 Spring Framework 的类对 JCL 包的 Log 对象都是可见 的。对于用户来说，所有 Spring 的版本都使用相同的日志包也是很重要的：因为保留了向 后兼容的特性，那么迁移是很容易进行的，同理，扩展 Spring 的应用程序也是这样。我们

这样做就是使得 Spring 中的模块明确地使用基于 commons-logging（JCL 的典型实现）的 日志实现，在编译时也使得其它模块都基于这个日志包。如果你正在使用 Maven 的话，同 时 想知道 在 哪儿 获 取 到的 commons-logging 依赖 ，那就是从 Spring 中被称作是 spring-core 的核心模块中获取的。

关于 commons-logging 比较好的做法是你不需要做其它的步骤就可以使应用程序运 行起来。它有一个运行时的查找算法，在我们都知道的类路径下寻找其它的日志框架，并且

使用 Spring 认为是比较合适的（或者告诉 Spring 你需要使用的具体是哪一个）一个。如果

没有找到可用的，那么你会得到一个来自 JDK（java.util.logging 或者简称为 JUL）本身看起来 还不错的日志依赖。那么，你会发现在很多时候，Spring 应用程序运行中会有日志信息打印 到控制台上，这点是很重要的。

1.3.2.1 不使用 Commons Logging

不幸的是，commons-logging 的运行时查找算法对最终用户方便是有些问题的。如果我们可以让时光倒流并让 Spring 从现在开始作为一个新的项目进行，那么我们会使用不 同的日志依赖。首选的日志依赖可能就是 Java 简单的日志门面（SLF4J），SLF4J 也被 Spring 的开发人员在他们其它应用程序中的很多工具所使用。

关闭 commons-logging 也很简单：仅仅要确认在运行时，它的包不在类路径下就可 以了。在 Maven 中要去掉这个依赖，因为 Spring 依赖声明的时候会带进来，那么就需要这 么来进行。

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

目前这个应用程序可能就不能运行了，因为在类路径中已经没有了 JCL API 的实现了， 所以要修复这个问题的话，就要提供另外一种日志的实现了。在下一节中，我们来说明如何 提供一个其它的 JCL 的实现，这里，我们使用 SLF4J 作为示例。

1.3.2.2 使用 SLF4J

SLF4J 是一个整洁的日志依赖，在运行时的效率也比 commons-logging 更高，因为

SLF4J 使用了编译时构建，而不是运行时去查找其它整合的日志框架。这也就意味着你可以

更明确地在运行时去做些什么，并且去声明或配置。SLF4J 为很多通用的日志框架提供的绑 定，所以通常你可以选择已有的一个，并去绑定配置或管理。

SLF4J 为很多通用日志框架提供绑定，包括 JCL，并且也可以反向进行：桥接其它的日志 框架和 SLF4J 本身。所以如果要在 Spring 中使用 SLF4J，那么就需要使用 SLF4J-JCL 桥来代替

commons-logging 依赖。只要配置好了，那么来自 Spring 的日志调用就会被翻译成调用

SLF4J 的 API 了，如果在应用程序中的其它类库使用了原有的 API，那么会有一个单独的地方 来进行配置和管理的日志。

一个常用的选择就是桥接 Spring 到 SLF4J API，之后提供从 SLF4J 到 Log4J 的明确绑定。 你需要提供 4 种依赖（并排除已经存在的 commons-logging 依赖）：桥接工具，SLF4J 的

API，绑定到 Log4J 的工具，还有 Log4J 本身的实现。那么在 Maven 的配置文件中，你就可 以这样来做：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
        <version>1.5.8</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.5.8</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.5.8</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

这样，看起来好像很多依赖都需要日志了。情况确实是这样，但 SLF4J 是可选的，而且 它的性能要比含有类加载器问题的 commons-logging 依赖好很多，尤其是如果你使用了 一个严格限制的容器，比如 OSGi 平台。据称那也会有性能优势，因为绑定是编译时进行的， 而不是在运行时。

另外，在 SLF4J 的用户中一个较为常见的选择是使用很少步骤，同时产生更少的依赖， 那也就是直接绑定到 Logback 上。这会去掉很多额外的绑定步骤，因为 Logback 本身直接实 现了 SLF4J ， 这 样 的 话， 你 就仅 需 要两个依赖的类库 ，而不 原先 的 是四 个 了（ 就是 jcl-over-slf4j 和 logback）。如果你也确实那么来做了，你可能还需要从其它外部依 赖（而不是 Spring）中去掉 slf4j-api 的依赖，因为在类路径中只保留一个 API 的一个版本就 行了。

1.3.2.3 使用 Log4J

很多用户出于配置和管理的目的而使用 Log4j 作为日志框架。这样也同样很有效率并且 易于创建，而且，Log4j 也是我们事实上在构建和测试 Spring 时，和运行时环境中使用的日 志框架。Spring 本身也提供一些工具来配置和初始化 Log4j，所以，在某些模块中它也提供 了可选的在编译时对 Log4j 框架的依赖。

要让 Log4j 框架和默认的 JCL 依赖（commons-logging）同时起作用，你所要做的就 是要将 Log4j 的类库 放到 类路 径下 ，并且 提供 配置 文件 （在类 路径 的根 路径 下放置 log4j.properties 或 log4j.xml 为名的文件）。而对于使用 Maven 的用户来说，下面 的代码可以是对依赖的声明：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.0.0.RELEASE</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

下面是 log4j.properties 打印到控制台的日志的配置示例：

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j .ConsoleAppender log4j.ap
pender.stdout.layout=org.apache.log4j.PatternLayout log4j.append
er.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L
- %m%n

log4j.category.org.springframework.beans.factory=DEBUG
```

运行时容器和本地的 JCL

很多用户在容器中运行 Spring 的应用程序，而该容器本身也提供了 JCL 的实现。比如 IBM 的 Webshphere 应用服务器（WAS）就是这种类型的。这通常就会引起问题，而不幸的是没 有什么解决的方法；很多情况下仅仅从应用程序中去掉 commons-logging 依赖是不够的。

我们要 清楚 地认 识这一 点： 这个 问题 通常和 JCL 本 身一同 报告 出来 ，或 者是和

commons-logging：而不是绑定 commons-logging 到另外的框架（这里，通常是 Log4J）。 这就可能会引发问题，因为 commons-logging 改变了它们在运行时环境里，在一些容器 中查找老版本（1.0）的方式，而现在很多人使用的是新版本（1.1）。Spring 不会使用任何不 通用的 JCL API 部分，所以这里不会有问题，但是 Spring 本身或应用程序尝试去进行日志记 录，你可以发现绑定到 Log4J 是不起作用的。

这种在 WAS 上的情况，最简单的做法就是颠倒类加载器的层次（IBM 称之为“parent last”），那么就是使应用程序来控制，而不是容器来控制 JCL 的依赖。这种选择并不总是开 放的，但是在公共领域中的替代方法也有其它的建议，根据确切的版本和容器的特性集，您所要求的效果可能会有不同。

第二部分 Spring 3 的新特性

第 2 章 Spring 3.0 的新特性和增强

如果你使用 Spring Framework，那么你会看到 Spring 现在主要有两个修订版本：一个是 在 2006 年 12 月发布的 Spring 2.0，还有一个是在 2007 年 11 月发布的 Spring 2.5。而现在是 第三个版本 Spring 3.0 了。

Java SE 和 Java EE 支持

Spring Framework 现在完全是基于 Java 5 和 Java 6 版本的。

此外，Spring 还兼容 J2EE 1.4 和 Java EE5，同时也会介绍一些对 Java EE6 的先期支持。

2.1 Java 5

框架的整体代码都已经修订到支持 Java 5 的新特性了，比如泛型，可变参数和其它的语 言方面的改进。我们也尽最大努力来保持代码的向后兼容性。现在我们也有一致的泛型集合 和 Map 的使用，一致的泛型 FactoryBean 使用，还有在 Spring AOP 的 API 中对桥接方法的一 致 性 解决方案。泛型 的 上 下文 监 听 器 仅 仅 自 动 接收 特 定 的 事 件 类 型。 所 有 的 比 如 TransactionCallback 和 HibernateCallback 回调接口现在也都声明为泛型返回值。总之，Spring 核心代码库都已经为 Java 5 而进行修订和优化了。

Spring 的 TaskExecutor 抽象也已经为和 Java 5 的 java.util.concurrent 的紧密整合而更新了。 我们现在为可调用特性提供顶级的类的支持，还有 ExecutorService 适配器，ThreadFactory 的整合等。这和 JSR-236（Java EE6 的并发工具）尽可能是一致的。此外，我们还提供对使 用新的@Async 注解（或者 EJB 3.1 的 @Asynchronous 注解）的异步方法调用的支持。

2.2 改进的文档

Spring 参考文档也有很大的更新，并反映出对 Spring 3.0 的改进和新特性。而每一项努 力都是来保证在文档中没有错误，但是多多少少还会存在一些错误。如果你确实发现了任意 的错字或者是严重的错误，你可以在午餐时间把它们画上圈，然后将这些错误报告给 Spring 的开发团队，你可以在 JIRA 上打开一个问题。

2.3 新的文章和教程

有很多优秀的文章和教程来展示如何开始使用 Spring 3 的新特性。您可以在
Spring 文档

页面来阅读它们。

示例代码也已经更新采用了 Spring 3 的新特性。此外，示例代码也已经从源码树中移入到了专用的 SVN 资源库中。SVN 资源库的访问地址是：https://anonsvn.springframework.org/svn/spring-samples/。

因此，这些示例代码就不会跟在 Spring 3 的发布包中了，需要你从上面提到的资源库中

分别去下载。而本文档会继续引用一些示例（特别是 Petclinic）来说明各种特性。注意

要获取关于 Subversion（简称为 SVN）的更多信息，可以参考 SVN 的项目主页，地址是：http://subversion.apache.org/。

2.4 新的模块组织方式和系统构建方式

框架的模块已经被重新修订了，现在的组织结构是：一个源码分支是一个 jar 文件：

- org.springframework.aop
- org.springframework.beans
- org.springframework.context
- org.springframework.context.support
- org.springframework.expression
- org.springframework.instrument
- org.springframework.jdbc
- org.springframework.jms
- org.springframework.orm
- org.springframework.oxm
- org.springframework.test
- org.springframework.transaction
- org.springframework.web
- org.springframework.web.portlet
- org.springframework.web.servlet
- org.springframework.web.struts

注意：

原来的 Spring.jar 文件包含了几乎所有框架内容，而现在不再提供了。

现在我们使用新的 Spring 构建系统，从熟知的 Spring Web Flow 2.0 中而来，它给了我们 如下的特性：

- 基于 Ivy 的“Spring 构建”系统
- 一致的开发过程
- 一致的依赖管理
- 一致的 OSGi 清单生成

2.5 新特性概述

下面列出的是 Spring 3.0 的新特性，我们在后面章节的详细讲述中会覆盖到这些特性。

- Spring 表达式语言
- IoC 增加/基于对 Java 的 bean 元数据的支持
- 通用的类型转换和字段格式化系统
- 对象转 XML 映射功能（OXM），这是从 Spring Web Service 项目中迁出的
- 全面的 REST 支持
- 增加@MVC
- 声明式的模型验证
- 先期对 Java EE 6 的支持
- 嵌入式数据库的支持

2.5.1 为 Java 5 更新的核心 API

BeanFactory 接口尽可能地返回该类型 bean 的实例：

- `T getBean(Class<T> requiredType)`
- `T getBean(String name, Class<T> requiredType)`
- `Map<String, T> getBeansOfType(Class<T> type)`

现在，Spring 的 TaskExecutor 接口扩展了 java.util.concurrent.Executor

- 扩展的 AsyncTaskExecutor 支持标准的可调用特性

新的基于 Java 5 的转换 API 和 SPI：

- 无状态的 ConversionService 和转换器
- 取代标准 JDK 的 PropertyEditors

泛型 `ApplicationListener<E>`

2.5.2 Spring 表达式语言

Spring 引入了一种表达式语言，这和统一的 EL 在语法上很相似，但是提供了很多新特 性。这种表达式语言可以用于定义基于 XML 和注解的 bean，也可以作为表达式语言的基础， 支持贯穿整个 Spring 框架的组合。这些新功能的详细内容可以在第 7 章 Spring 表达式语言（SpEL）中看到。

Spring 表达式语言被用来为 Spring 社区提供一个单一的，支持良好的表达式语言，可以 用于 Spring 系列的所有产品中。它的语言特性由 Spring 系列产品的所有项目的需求来共同 驱动，包括基于 Eclipse 的 SpringSource Tool Suite（SpringSource 组织开发的工具套件，译者 注）代码完成支持工具的需求。

下面是表达式语言如何被用于配置数据库属性设置的示例代码：

```
<bean class="mycompany.RewardsTestDatabase">
    <property name="databaseName" value="#{systemProperties.data
baseName}" />
    <property name="keyGenerator"
        value="#{strategyBean.databaseKeyGenerator}"/>
</bean>
```

如果你使用注解来配置组件，那么这个功能也是可用的：

```
@Repository
public class RewardsTestDatabase {
    @Value("#{systemProperties.databaseName}")
    public void setDatabaseName(String dbName) { … }
    @Value("#{strategyBean.databaseKeyGenerator}" )
    public void setKeyGenerator(KeyGenerator kg) { … }
}
```

2.5.3 控制反转（IoC）容器

2.5.3.1 基于 Java 的 bean 元数据

从 Java Config 项目中引入的一些核心特性现在也被加入到 Spring Framework 中了。这就 是说下面的注解是直接被支持的。

- @Configuration
- @Bean
- @DependsOn
- @Primary
- @Lazy
- @Import
- @ImportResource
- @Value

下面是一个 Java 类提供基本配置信息的示例，使用了新的 Java Config 特性：

```
package org.example.config;
@Configuration
public class AppConfig {
    private @Value("#{jdbcProperties.url}") String jdbcUrl;
    private @Value("#{jdbcProperties.username}") String username
;
    private @Value("#{jdbcProperties.password}") String password
;
    @Bean
    public FooService fooService() {
        return new FooServiceImpl(fooRepository());
    }
    @Bean
    public FooRepository fooRepository() {
        return new HibernateFooRepository(sessionFactory());
    }
    @Bean
    public SessionFactory sessionFactory() {
        // 装配一个session factory
        AnnotationSessionFactoryBean asFactoryBean =
        new AnnotationSessionFactoryBean(); asFactoryBean.setDat
aSource(dataSource());
        // 其它配置
        return asFactoryBean.getObject();
    }
    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(jdbcUrl, username, pa
ssword);
    }
}
```

要让它发挥作用，你需要在应用上下文的 XML 文件中添加如下的组件扫描项。

```
<context:component-scan base-package="org.example.config"/>
<util:properties id="jdbcProperties" location="classpath:org/exa
mple/config/jdbc.properties" />
```

或者你可以直接使用 AnnotationConfigApplicationContext 来启动被 @Configuration 注解的类：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationCont
ext(AppConfig. class);
    FooService fooService = ctx.getBean(FooService.class); fooSe
rvice.doStuff();
}
```

参考 4.12.2 节“使用 AnnotationConfigApplicationContext 实例化 Spring 容器”来参考关 于 AnnotationConfigApplicationContext 的全部内容。

2.5.3.2 使用组件定义 bean 的元数据

@Bean 注解的方法也可以支持内部的 Spring 组件。它们贡献工厂 bean 的定义到容器中。 参考 4.10.4 节“使用组件定义 bean 的元数据”来获取更多信息。

2.5.4 通用的类型转换系统和字段格式化系统

通用的类型转换系统（参考 6.5 节）已经引入了。系统现在被 SpEL 使用来进行类型转 换，当绑定 bean 的属性值时也会被 Spring 容器和数据绑定器使用。

此外，格式化（参考 6.6 节）SPI 也被引入了，来格式化字段值。这个 SPI 对 JavaBean 的 PropertyEditors 提供了简单的，更强壮替代，在如 Spring MVC 的客户端环境中来使用。

2.5.5 数据层

现在，对象到 XML 映射功能（OXM）被从 Spring Web Service 项目中移到 Spring Framework 的核心中。这个功能可以在 org.springframework.oxm 包下找到。关于使用 OXM 模块 的更多信息可以在第 15 章使用 O/X 映射器编组 XML 找到。

2.5.6 Web 层

对于 Web 层来说，最令人兴奋的新特性是对构建 RESTful Web Service 和 Web 应用程序 的支持。也有一些新的注解可以用于任意的 Web 应用程序。

2.5.6.1 全面的 REST 支持

对构建 RESTful 应用程序服务器端的支持已经作为已有的注解驱动的 MVC web 框架而提 供了。客户端的支持由 RestTemplate 类来提供，和其 它的模板类是相 似的，比如 JdbcTemplate 和 JmsTemplate。服务器和客户端两者的 REST 功能都使用 HttpConverter 来在对象和它们在 HTTP 请求和响应代表之间方便的转换。

MarshallingHttpMessageConverter 使用之前提到的对象到 XML 映射功能。 可以参考 MVC（第 16 章）和 RestTemplate（20.9.1 节）部分获取更多信息。

2.5.6.2 @MVC 的增加

mvc 命名空间被引入来大大简化 Spring MVC 的配置。

其它如 @CookieValue 和 @RequestHeader 的 注 解 也 被 加 入 了 。 参 考 使 用

@CookieVlue 注解映射 cookie 值（16.3.3.9 节）和使用@RequestHeader 注解映射请求头部属 性（16.3.3.10 节）来获取更多信息。

2.5.7 声明式的模型验证

一些验证增强（6.7 节），包括 JSR303 支持，使用 Hibernate 校验器作为默认提供者。

2.5.8 先期对 Java EE 6 的支持

通过使用新的@Async 注解（或 EJB 3.1 的@Asynchronous 注解）我们提供异步方法调用。

JSR 303，JSF 2.0，JPA 2.0 等

2.5.9 嵌入式数据库的支持

现在也提供了对嵌入式 Java 数据库引擎（13.8 节）的方便支持，包括 HSQL，H2 和 Derby。

第 3 章 Spring 3.1 的新特性和增强

在 Spring 3.0 引入的支持之上构建，Spring 3.1 现在还在开发中，目前 Spring 3.1 M2 才刚 刚发布。

3.1 新特性概述

（该部分内容待 Spring 3.1 GA 发布后翻译）

第三部分 核心技术

参考文档的这一部分涵盖了 Spring Framework 中不可或缺的技术。 这些内容最主要的是 Spring Framework 的控制反转（IoC）容器。Spring Framework 的 IoC

容器的完全使用是紧跟其后的 Spring 的面向切面编程（AOP）技术的完全覆盖。Spring Framework 有它自己的 AOP 框架，在概念上很容易去理解，在 Java 企业级编程中，它成功地解决了 80%的 AOP 需求的功能点。

也提供了涵盖的 Spring 和 AspectJ（目前最丰富的 - 在功能方面 – 当然是在 Java 企业 级空间中最成熟的 AOP 实现）的整合。

最终，通过测试驱动开发（test-driven-development，TDD）的软件开发方法，也是 Spring 团队所主张的，所以 Spring 对整合测试的支持也涵盖到了（沿袭单元测试的最佳实践）。Spring 团队也发现了 IoC 的正确使用，当然，这会让单元和集成测试更容易（setter 方法的存在和 类的适当的构造方法可以使得它们很容易的在测试时连接在一起，而不需要设立服务定位器注册和诸如此类的方法）。这章专门的测试又往说服你。

- 第 4 章，IoC 容器
- 第 5 章，资源
- 第 6 章，验证，数据绑定和类型转换
- 第 7 章，Spring 表达式语言（SpEL）
- 第 8 章，使用 Spring 进行面向切面编程
- 第 9 章，Spring 的 AOP API
- 第 10 章，测试

第 4 章 IoC 容器

4.1 Spring IoC 容器和 bean 的介绍

本章涵盖了 Spring Framework 的控制反转容器（IoC）[参考 1.1 节的背景]原则的实现。 IoC 也被称为是依赖注入（DI）。这是一个对象定义它们依赖的过程，也就是说，它们使用的 其它对象，仅仅通过构造方法参数，工厂方法参数或在对象被创建后的实例上设置的属性， 亦或者是从工厂方法返回的参数。之后容器在它创建 bean 的时候注入那些依赖。这个过程 是根本上的反向，因此名称是控制反转（IoC），bean 本身控制实例化或直接地使用类的构 造来定位它的依赖，或者是如服务定位器模式的机制。

org.springframework.beans 和 org.springframework.context 包是 Spring Framework 的 IoC 容器的基础。BeanFactory 接口提供高级的配置机制，可以管理任意类 型的对象。ApplicationContext 是 BeanFactory 的子接口。它添加了和 Spring 的 AOP 特性很简便的整合；消息资源处理（用于国际化 i18n），事件发布；应用层特定的上下文， 比如用于 Web 应用程序的 WebApplicationContext。

总之，BeanFactory 提供了配置框架和基本功能，而 ApplicationContext 添加 了更多企业级开发特定的功能。ApplicationContext 是 BeanFactory 完整的超集， 专门用于本章，来描述 Spring 的 IoC 容器。对于使用 BeanFactory 而不是 ApplicationContext 的更多信息，可以参考 4.15 节“BeanFactory”。

在 Spring 中，对象构成应用程序的骨感，它们是由 Spring 的 IoC 容器管理的，并被称为 bean。一个 bean 就是一个实例化并组装的对象，由 Spring 的 IoC 容器来管理。否则，bean 就是应用程序中众多对象之一。Bean 和它们之间的依赖，反射出由容器使用的配置元数据。

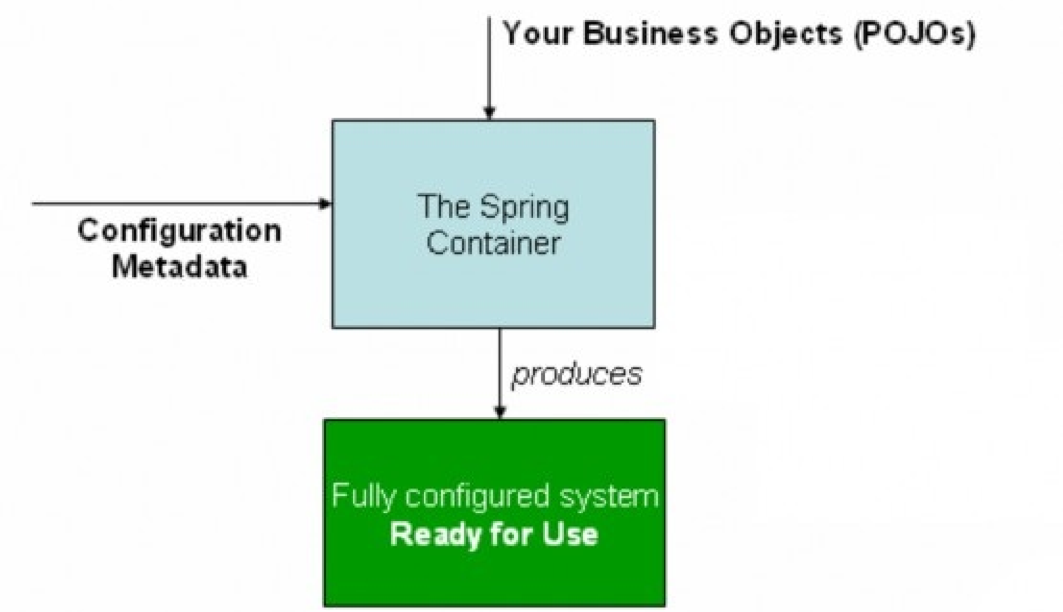
4.2 容器概述

org.springframework.context.ApplicationContext 接口代表了 Spring 的 IoC 容器，负责实例化，配置和装配上述的 bean。容器获得指示来实例化某对象，配置并装配，这都是通过读取配置元数据实现的。配置元数据在 XML 中，Java 注解或 Java 代码中来 表示。它允许你表达编写应用程序的对象，还有对象间丰富的相互依存的关系。

ApplicationContext 接口的一些实现使用 Spring 开箱的支持。在独立的应用程序 中，通 常 是 来 创 建 ClassPathXmlApplicationContext 或 FileSystemXmlApplicationContext 的实例。XML 是定义配置元数据的传统格式，你 可以指示容器使用 Java 的注解或是代码作为元数据的格式，提供少量的 XML 配置声明来开 启对这些额外的元数据格式的支持。

在很多应用场景中，明确的用户代码不需要实例化一个或者多个 Spring IoC 容器的实例。 比如，在 Web 应用场景中，在应用程序的 web.xml 中简单的八（左右）行样板 J2EE 描述 符 XML 文件通常就够了（参考 4.14.4 节"对 Web 应用程序方便的应用上下文实例化"）。如 果你正使用 SpringSource 的工具套件，Eclipse 支持的开发环境或者是 Spring ROO 这样的样板 配置，就可以容易地被创建，点几下鼠标或按键就可以了。

下图是 Spring 如何工作的高级别视图。你的应用程序类联合配置元数据，所以在 ApplicationContext 被创建和实例化后，就得到了一个完全配置的可执行系统或程序。



Spring 的 IoC 容器

4.2.1 配置元数据

正如上图所示，Spring 的 IoC 容器处理配置元数据的一种形式；这个配置元数据代表了 你作为应用开发人员是如何告诉 Spring 容器在你的应用程序中来实例化，配置并装配对象 的。

配置元数据传统上是以直观的 XML 格式提供的，这是本章的大部分内容使用它来传达 Spring IoC 容器关键概念和功能。

> 注意
>
> 基于 XML 的元数据并不是唯一的配置元数据格式。这种配置元数据眞正写入时，Spring 的 IoC 容器本身和这种格式完全脱钩。

关于 Spring 容器使用元数据格式的信息，可以参考：

基于注解的配置（4.9 节）：Spring 2.5 引入了对基于注解元数据的支持。

基于 Java 的配置（4.12 节）：从 Spring 3.0 开始，很多由 Spring JavaConfig 项目提供的特 性称为 Spring Framework 的核心。因此你可以在应用程序外部来定义 bean，使用 Java 代码而不是 XML 文件。要使用这些新的特性，请参考 @Configuration，@Bean，

@Import 和@DependsOn 注解

Spring 配置典型的是最少由一个容器必须管理的 bean 定义构成。基于 XML 的配置元数据展示了这些 bean 的配置是用在顶级 `<beans/>` 元素中的 `<bean/>` 元素完成的。

这些 bean 的定义对应构成应用程序中眞实的对象。比如你定义的服务层的对象，数据访问对象（Data Access Object，DAO），表示对象比如 Struts 的 Action 实例，基础设置对象 比如 Hibernate 的 SesstionFactories，JMS 的 Queues 等这些典型的例子。而不用在容 器中定义细粒度的领域模型对象，因为这通常是由 DAO 和业务逻辑负责创建并加载的领域 对象。但是你也可以使用 Spring 和 AspectJ 整合来配置创建在 IoC 容器控制之外的对象。参 考使用在 Spring 中使用 AspectJ 来对领域对象进行依赖注入（8.8.1 节）。

下面的示例展示了基本的基于 XML 的配置元数据的结构：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema -instance"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
    http://www.springframework.org/schema/beans/spring -beans-3.
0.xsd">
    <bean id="..." class="...">
        <!-- 这个bean的合作者和配置在这里编写 -->
    </bean>
    <bean id="..." class="...">
        <!-- 这个bean的合作者和配置在这里编写 -->
    </bean>
    <!-- 更多的bean定义在这里编写 -->
</beans>
```

id 属性是一个字符串，用来标识定义的独立的 bean。class 属性定义了 bean 的类型， 需要使用类的完全限定名。id 属性的值指的就是协作对象。指写作对象的 XML 在这个示例 中没有展示；参考依赖（4.4 节）来获取更多信息。

4.2.2 实例化容器

实例化 Spring 的 IoC 容 器 是 很 简 单 的 。 定 位 路 径 或 所 有 路 径 提 供 给 ApplicationContext 的构造方法，实际上是表示资源的字符串，它就允许容器从各种外 部资源比如本地文件系统，Java 的 CLASSPATH 等来加载配置元数据

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.x
ml",
    "daos.xml"});
```

> ⓘ
>
> 注意
>
> 在学习过 Spring 的 IoC 容器之后，你可能想更多了解 Spring 的 Resource 抽象，这在 第 5 章，资源中会描述，它提供了一个从定义 URI 语法的位置读取输入流的简便机制。特别 是，Resource 路径用来构建应用程序上下文，这会在 5.7 节“应用上下文和资源路径”中 来描述。

下面的示例展示了服务层代码（services.xml）的配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
    http://www.springframework.org/schema/beans/spring -beans-3.
0.xsd">
    <!-- services -->
    <bean id="petStore"
        class="org.springframework.samples.jpetstore.services
        .PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- 这个bean的其它合作者和配置在这里编写-->
    </bean>
    <!-- 更多的service bean的定义在这里编写 -->
</beans>
```

下面的示例展示了数据访问对象的 daos.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
    http://www.springframework.org/schema/beans/spring -beans-3.
0.xsd">
    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.ibat is
.SqlMapAccountDao">
        <!-- 这个bean的其它合作者和配置在这里编写 -->
    </bean>
    <bean id="itemDao"
        class="org.springframework.samples.jpetstore.dao.ibat is
.SqlMapItemDao">
        <!-- 这个bean的其它合作者和配置在这里编写 -->
    </bean>
    <!-- 更多数据访问对象的定义在这里编写 -->
</beans>
```

在上面的示例中，服务层代码由类 PetStoreServiceImpl 和两个基于 iBatis（iBatis 现已更名为 MyBatis，译者注）对象/实体映射框架的数据访问对象 SqlMapAccountDao 和 SqlMapItemDao 构成。property 的 name 元素指的是 JavaBean 的属性名，而 ref 元 素指的是其它 bean 定义的名称。id 和 ref 元素之间的这种联系表示了两个协作对象的依赖 关系。要了解更多配置对象依赖的信息，可以参考依赖（4.4 节）。

4.2.2.1 处理基于 XML 的配置元数据

跨越多个 XML 文件中定义 bean 是很有用的。通常每个独立的 XML 配置文件代表了一 个逻辑层或架构中的一个模块。

你可以使用 ApplicationContext 的构造方法从所有的 XML 文件片段中来加载 bean。这个 构造方法可以接收多个 Resource 位置，这在之前的部分都已经看到了。另外，可以使用一个或多个 `<import/>` 元素来从另外的一个或多个文件中加载 bean。比如：

```
<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>
    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

在这个示例中，外部的 bean 通过三个文件来加载，分别是 services.xml，messageSource.xml 和 themeSource.xml。所有的位置路径都是相对于该引用文件的，所以 service.xml 必须在和引用文件相同路径中或者是类路径下，而 messageSource.xml 和 themeSource.xml 必须是在位于引用文件下一级的 resources 路径下。正如你看到的，前部的斜杠被忽略了，这是由于路径都是相对的，最 好就不用斜线。文件的内容会被引入，包括顶级的 `<beans/>` 元素，根据 Spring 的 Schema 或 DTD，它必须是有效 bean 定义的 XML 文件。

> 注意
>
> 在父目录中使用相对路径“../”来引用文件，这是可能的，但是不推荐这么做。这么做了会创建一个文件，它是当前应用程序之外的一个依赖。 特别是，这种引用对于“classpath：”的 URL（比如，“classpath:../service.xml”）是不推荐的，运行时的解析过 程会选择“最近的”类路径根目录并且会查看它的父目录。类路径配置的修改可能会导 致去选择一个不同的，不正确的目录。 你也可以使用资源位置的完全限定名来代替相对路径：比如，“file: C:/config /services.xml”或“classpath:/config/services.xml”。这样的话，要注意你会耦合应用程序的配置到指定 的绝对路径。对于绝对路径，一般最好是保持一个间接的使用，比如通过占位符“${...}”， 这会基于运行时环境的 JVM 系统属性来解决。

4.2.3 使用容器

ApplicationContext 是能维护不同的 bean 和它们依赖注册的高级工厂接口。使用 `T getBean(Stringname, Class<T> requiredType)` 方法你可以获取 bean 的实例。

ApplicationContext 允许你读取 bean 并且访问它们，如下所示：

```
// 创建并配置bean
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.x
ml",
    "daos.xml"});
// 获取配置的实例
PetStoreServiceImpl service = context.getBean("petStore", PetSto
reServiceImpl.class);
// 使用配置的实例
List userList service.getUsernameList();
```

使用 getBean()方法来获取 bean 的实例。ApplicationContext 接口有一些其它方法来获取 bean，但最好应用程序代码不使用它们。事实上，应用程序代码应该没有 getBean()方法的调用，那么就没有对 Spring API 的依赖了。比如，Spring 和 Web 框架整 合时，提供了对各种 Web 框架类库的依赖注入，不如控制器和 JSF 管理的 bean。

4.3 Bean 概述

Spring 的 IoC 容器管理一个或多个 bean。这些 bean 通过提供给容器的配置元数据被创 建出来，比如，在 XML 中的 `<bean/>` 定义的形式。

在容器本身，这些 bean 代表了 BeanDefinition 对象，它们包含（在其它信息中）下列元数据：

打包的类限定名：就是这些 bean 的真正实现类。

bean 的行为配置元素，这表示了 bean 在容器中（范围，生命周期回调等等）应该是怎 样的行为。

对其它 bean 的引用，这是该 bean 工作所需要的；这些引用通常被称为合作者或依赖。

在新被创建的对象中的其它配置设置，比如，管理连接池的 bean 中使用的连接数或连 接池限制的大小。

元数据翻译成一组属性集合来构成每一个 bean 的定义。

表 4.1 bean 定义

属性	解释章节
class	4.3.2 节，“实例化 bean”
name	4.3.1 节，“命名 bean”
scope	4.5 节，“bean 的范围”
constructor arguments	4.4.1 节，“依赖注入”
properties	4.4.1 节，“依赖注入”
autowiring mode	4.4.5 节，“装配合作者”
lazy-initialization mode	4.4.4 节，“延迟初始化 bean”
initialization method	4.6.1.1 节，“初始化回调”
descruction method	4.6.1.2 节，“销毁回调”

此外，bean 的定义还包含如何创建特定 bean 的信息，ApplicationContext 的实现类允许用户将容器外部创建的已有对象的注册。这可以通过 getBeanFactory()方法访问 ApplicationContext 的 BeanFactory 来完成，它会返回 BeanFactory 的实现类 DefaultListableBeanFactory。DefaultListableBeanFactory 通过 registerSingleton(..)和 registerBeanDefinition(..)方法来支持这种注册。而典型的应用程序只和通过元数据定义的 bean 来工作。

4.3.1 命名 bean

每个 bean 有一个或者多个标识符。这些标识符必须在托管 bean 的容器内唯一。通常 一个 bean 只有一个标识符，但如果它需要多个的话，其它的可以被认为是别名。

在基于 XML 的配置元数据中，你可以使用 id 和/或 name 属性来指定 bean 的标识符。 id 属性允许你指定一个准确的 id。按照管理这些名称是字母和数字（‘myBean’，‘fooService’ 等），但是可能是特殊字符。如果你想为 bean 引入别名，你也可以在 name 属性中来指定，

通过逗号（,），分号（;）或空格来分隔。作为对历史的说明，在 Spring 3.1 版之前，id 属性 是 xsd:ID 类型，只限定为字符。在 3.1 版本中，现在是 xsd:string 类型了。要注意 bean 的 id 的唯一性还是被容器所强制的，但是对于 XML 处理器却不是。

当然你也可以不给 bean 提供 name 或 id。如果没有提供明确的 name 或 id，那么容器会为这个 bean 生成一个唯一的名称。然而，如果你想通过名称来参考这个 bean，那么可以 使用 ref 元素或者是服务定位器（4.15.2 节）风格的查找，你必须提供一个名称。不提供名称的动机是和使用内部 bean（4.4.2.3 节）或自动装备合作者（4.4.5 节）相关的。

Bean 的命名约定

该约定是用于当命名 bean 时，对实例字段名称的标准 Java 约定。也就是说，bean 的名称以小写字母开始，后面是驼峰形式的。这样的命名可以是（没有引号）‘accountManager’，‘accountService’，‘userDao’，‘loginController’ 等。

一致的命名 bean 可以使你的配置信息易于阅读和理解，如果你使用 Spring 的 AOP， 当应用通知到一组相关名称的 bean 时，它会给你很大的帮助。

4.3.1.1 在 bean 定义外面起别名

在 bean 定义的本身，你可以为 bean 提供多于一个的名称，使用一个由 id 属性或 name 属性中的任意名称数指定的名称组合。这些名称对于同一个 bean 来说都是相等的别名，在 一些情况下，这是很有用的，比如在应用程序中允许每个组件参考一个共同的依赖，可以使 用为组件本身指定的 bean 的名称。

然而，在 bean 定义时指定所有的别名是不够的。有事可能想在任意位置，为一个 bean 引入一个别名。这在大型系统中是很常见的例子，其中的配置信息是分在每个子系统中的， 每个子系统都有它自己的对象定义集合。在基于 XML 的配置元数据中，你可以使用 `<alias/>` 元素来完成这项工作。

```
<alias name="fromName" alias="toName"/>
```

在这个示例中，在相同容器中的名称为 fromName 的 bean，在使用过这个别名定义后， 也可以使用 toName 来指引。

比如，在子系统的配置元数据中，A 可能要通过名称‘subsystemA-dataSource’来指向 数据源。为子系统 B 的配置元数据可能要通过名称‘subsystemB-dataSource’来指向数据源。

当处理使用了这两个子系统的主程序时，主程序要通过名称‘myApp-dataSource’来指向数 据源。那么就需要三个名称指向同一个对象，那么就要按照下面的别名定义来进行添加

MyApp 的配置元数据：

```
<alias name="subsystemA-dataSource" alias="subsystemB-dataSource
"/>
<alias name="subsystemA-dataSource" alias="myApp-dataSource" />
```

现在每个组件和主程序都可以通过一个唯一的保证不冲突的名称来还有其它任意定义（更有效地是创建命名空间）来参照数据源了，当然它们参照的是同一个 bean。

4.3.2 实例化 bean

bean 的定义信息本质上是创建一个或多个对象的方子。当需要一个 bean 时，容器查找 这些方子来查找命名的 bean，并且使用由 bean 定义信息封装的配置元数据来创建（或获得） 一个真实的对象。

如果你使用基于 XML 的配置元数据，你为要被实例化的对象所指定的类型（或类）是 `<bean/>` 元素中 class 属性。这个 class 属性，是 BeanDefinition 实例内部的 cla ss 属性，通常是强制要有的。（对于特例，可以参考 4.3.2.3 节，“使用实例的工厂方法来实例化”和 4.7 节，“Bean 定义的继承”）你可以以两种方法之一来使用 class 属性：

典型的是，指定要被构造的 bean 类，容器本身直接通过反射调用它的构造方法来创建bean，也就是和 Java 代码中使用 new 操作符是相同的。

指定包含 static 工厂方法的真实的类，会被调用来创建对象，在一些不太常见的情况下，容器会调用类的 static 工厂方法来创建 bean。从调用 static 工厂方法返回的对象类型可能 是和另外一个类完全相同的类。

内部类名称

如果你想为 static 嵌入的类配置 bean，你需要使用内部类的二进制名称。 比如，如果在 com.example 包下有一个 Foo 类,而这个 Foo 类有一个 static 的内部类 Bar，那么在定义 bean 时’class’属性的值就会是...

```
com.example.Foo$Bar
```

请注意名称中$符号的使用，用来从外部类名中分隔内部类名。

4.3.2.1 使用构造方法实例化

当你使用构造方法来创建 bean 时，所有普通的类的使用都和 Spring 兼容。也就是说， 开发中的 bean 不需要实现任何特定的接口或以特定的方式来编码。仅简单指定 bean 的类 就足够了。但基于你使用的是什么类型的 IoC，就可能需要一个默认（空的）构造方法。

Spring 的 IoC 容器可以虚拟地管理任意的你想让它管理的类；而不仅仅限于管理真正的 JavaBean。很多 Spring 用户喜欢在容器中使用有默认（无参数）构造方法和在其后有适当 setter 和 getter 方法的真正 JavaBean。你也可以在容器中使用很多异样的，非 bean 样式的类。比 如，需要使用遗留的连接池，但是它没有符合 JavaBean 的规范，Spring 也能照样管理它。

基于 XML 的配置元数据，你可以如下来定义 bean：

```
<bean id="exampleBean" class="examples.ExampleBean"/>
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

关于提供构造方法参数（如果需要）和在对象被构造后，设置对象实例属性机制的详情， 请参考 4.4.1 节依赖注入。

4.3.2.2 使用静态工厂方法来实例化

当使用静态工厂方法来定义 bean 的时候，可以使用 class 属性来指定包含 static 工厂 方法的类，而名为 factory-method 的属性来指定静态方法。你应该调用这个方法（还可 以有可选的参数）并返回一个实际的对象，随后将它视为是通过构造方法创建的一样。在遗留代码中，这样定义 bean 的使用方式之一是调用 static 工厂。

下面 bean 的定义指定了要通过调用工厂方法生成的 bean。这个定义没有指定返回对象 的类型（类），仅仅是包含工厂方法的类。在本例中，createInstance()方法必须是静 态方法。

```
<bean id="clientService" class="examples.ClientService" factory-
method="createInstance"/>
public class ClientService {
    private static ClientService clientService = new
        ClientService();
    private ClientService() {}
    public static ClientService createInstance() {
        return clientService;
    }
}
```

关于提供（可选的）参数到工厂方法并在对象由工厂返回后设置对象实例属性的机制详 情，请参考 4.4.2 节深入依赖和配置。

4.3.2.3 使用实例工厂方法来实例化

和使用静态工厂方法（4.3.2.2 节）实例化相似，使用实例工厂方法实例化是要调用容器 中已有 bean 的一个非静态的方法来创建新的 bean。要使用这个机制，请把 class 属性留 空，但在 factory-bean 属性中指定当前（或父/祖先）容器中 bea n 的名字，该 bean 要 包含被调用来创建对象的实例方法。使用 factory-method 方法来设置工厂方法的名称。

```
<!-- 工厂bean，包含名为createInstance()的方法 -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator"
>
    <!-- 为locator bean注入任意需要的依赖 -->
</bean>
<!-- 通过工厂bean来创建的bean -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
public class DefaultServiceLocator {
    private static ClientService clientService = new
        ClientServiceImpl();
    private DefaultServiceLocator() {}
    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

一个工厂类也可以有多于一个工厂方法，比如下面这个：

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator"
>
    <!--为locator bean注入任意需要的依赖 -->
</bean>
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
<bean id="accountService" factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
public class DefaultServiceLocator {
    private static ClientService clientService = new
    ClientServiceImpl();
    private static AccountService accountService = new
    AccountServiceImpl();
    private DefaultServiceLocator() {}
    public ClientService createClientServiceInstance() {
        return clientService;
    }
    public AccountService createAccountServiceInstance() {
        return accountService;
    }
}
```

这个方法展示了工厂 bean 本身可以通过依赖注入（DI）被管理和配置。请参考 4.4.2 节

深入依赖和配置。 注意在 Spring 文档中，工厂 bean 指的是在 Spring 容器中配置的 bean，可以通过实例（4.3.2.3节）或静态（4.3.2.2 节）工厂方法来创建对象。与此相反的是，FactoryBean（注意大小写）指的是 Spring 特定的 FactoryBean（4.8.3 节）

4.4 依赖

典型的企业级应用程序不是由单独对象（或 Spring 中的 bean）构成的。尽管最简单的 应用程序有一些对象协同工作来表示终端用户所看到的连贯的应用。下一节会解释如何去为 完全实现的应用程序定义一组独立的 bean，其中对象间相互协作来达到目标。

4.4.1 依赖注入

依赖注入（DI）是对象定义它们依赖的过程，也就是说，要和它们协同工作其它对象， 仅仅可以通过构造方法参数，工厂方法参数，或者是在工厂方法返回的对象或被构造好后，

为对象实例设置的属性。容器当创建好 bean，随后就会注入那些依赖。这个过程从根本上 来说是反向的，因此命名为控制反转（IoC），bea n 本身直接使用构造好的类或服务定位器 模式来控制实例或它的依赖的所在位置。

应用了 DI 原则，代码就干净多了，当为对象提供它们依赖的时候，解耦是很有效率的。

对象不再去检查它的依赖，也不需要知道位置或依赖的类。因此，类就很容易被测试，特别 是当依赖是接口或抽象基类的时候，这允许在单元测试中使用 stub 或 mock 的实现类。

DI 存在两种主要的方式，基于构造方法的依赖注入（4.4.1.1 节）和基于 setter 方法的依赖注入（4.4.1.2 节）。

4.4.1.1 基于构造方法的依赖注入

基于构造方法的依赖注入是容器调用构造方法和一组参数完成的，每个都表示着一个依 赖。使用特定的参数来调用 static 工厂方法构造 bean 基本也是相同的，这种说法把给构 造方法的参数和给 static 工厂方法的参数相类似。下面的示例展示了一个仅使用构造方 法进行依赖注入的饿类。注意这个类没有什么特殊之处，就是一个 POJO 而且没有对容器特 定接口，基类或注解的依赖。

```
public class SimpleMovieLister {
    // SimpleMovieLister对MovieFinder有依赖
    private MovieFinder movieFinder;
    // 这个构造方法使得Spring容器可以’注入’MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // 业务逻辑就可以’使用’注入的MovieFinder了，代码就省略了...
}
```

构造方法参数解析

构造方法参数解析匹配使用参数的类型。如果在 bean 的构造方法参数不存在潜在的歧义，那么当 bean 被实例化的时候，定义的构造方法参数的顺序就是被提供到适当构造方法 参数的顺序。请看下面的类：

```
package x.y;
public class Foo {
    public Foo(Bar bar, Baz baz) {
        // ...
    }
}
```

没有潜在的歧义存在，假设 Bar 和 Baz 类没有继承关系。因此下面的配置就可以使用 了，而且并不需要明确地在 `<constructor-arg/>` 元素中去指定构造方法参数的索引和/ 或类型。

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
    </bean>
    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>
</beans>
```

当另外一个 bean 被引用时，类型是明确的，那么匹配就能成功（前面示例中也是这样 的）。当使用简单类型时，比如 `<value>true<value>` ，Spring 不能决定值的类型，所以 没有帮助是不能匹配的。看下面的示例：

```
package examples;
public class ExampleBean {
    //计算最佳答案的年数
    private int years;
    // 生命，宇宙，所有问题的答案
    private String ultimateAnswer;
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

构造方法参数类型匹配

在上面的情形下，如果你使用 type 属性明确地为构造方法参数指定类型的话，容器可 以进行匹配简单的类型。比如：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

构造方法参数索引

使用 index 属性来指定明确的构造方法参数索引。比如：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

此外，为了解决多个简单值的歧义，如果构造方法有两个相同类型的参数时，指定所以 可以解决歧义。注意索引是基于 0 开始的。

构造方法参数名称

在 Spring 3.0 中，你也可以使用构造方法参数名称来消除歧义：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateanswer" value="42"/>
</bean>
```

要记住要使得这种方式可用，代码必须和内嵌的调试标识一起编译，那样 Spring 才可 以从构造方法中来查找参数。如果没有和调试标识（或不想）一起编译，那么可以使用 JDK 的注解@ConstructorProperties 来明确构造方法参数。那么示例代码就如下所示：

```
package examples;
public class ExampleBean {
    // 忽略属性
    @ConstructorProperties({"years", "ultimateAnswer"})
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

4.4.1.2 基于 setter 方法的依赖注入

基于 setter 方法的依赖注入由容器在调用过无参数的构造方法或无参数的 static 工厂 方法来实例化 bean 之后，再调用 bean 的 setter 方法来完成的。

下面的示例展示了一个仅仅能使用纯 setter 方法进行依赖注入的类。这是一个常见的 Java 类，是一个没有依赖容器指定的接口，基类或注解的 POJO。

```
public class SimpleMovieLister {
    // SimpleMovieLister对MovieFinder有依赖
    private MovieFinder movieFinder;
    // setter方法可以让Spring容器来'注入'MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // 真正'使用'注入的MovieFinder的业务逻辑代码被省略了...
}
```

ApplicationContext 对它管理的 bea n 支持基于构造方法和 setter 方法的依赖注入。

也支持在一些依赖已经通过构造方法注入之后再进行 setter 方法注入。以

BeanDefinition 形式来配置依赖，使用 PropertyEditor 实例将属性从一种形式格式化到另一种。但很多 Spring 的用户不直接（编程时）使用这些类，而是使用 XML 文件来定 义，之后会在内部转换这些类的实例，需要加载整个 Spring IoC 容器的实例。

基于构造方法还是 setter 方法进行依赖注入？ 因为你可以将二者混淆，那么对于基于构造方法或是 setter 方法的依赖注入，有一个很好的规则就是为强制依赖使用构造方法参数，而对于可选参数使用 setter 方法。要注意在 setter 方法上使用注解 @Required（4.9.1 节），可用于 setter 方法所需的依赖。

Spring 团队通常主张使用 setter 方法注入，因为大量的构造方法参数会使程序变得非 常笨拙，特别是当属性为可选的时候。Setter 方法会让该类的对象今后适合于重新配置或 重新注入。通过 JMX Mbean（第 23 章）来管理就是一个令人关注的用例。

一些人纯粹赞成构造方法注入。提供所有对象的依赖意味着对象在完全初始化状态时，通常要返回客户端（调用）代码。这个缺点会使得对象变得不适合去重新配置或重新注入。

使用依赖注入的时候要特别注意一些类。当要选择处理没有源代码时的第三方类库的时候。遗留的类可能没有暴露任何 setter 方法，而构造方法注入则是唯一可用的依赖 注入方式。

4.4.1.3 解决依赖过程

容器按如下步骤来解决 bean 的依赖：

1. ApplicationContext 和描述了所有 bean 的配置元数据一起被创建并初始化。配置元数据可以通过 XML，Java 代码或注解来指定。

2. 对于每一个 bean 来说，它的依赖被表述为属性，构造方法参数的形式，如果你使用 了静态工厂方法来代替构造方法，那么还会是静态工厂方法参数的形式。当 bean 被实际创 建时，这些依赖被提供给 bean。
3. 每个属性或构造方法参数就是一个要被设置的值或者是容器中其它 bean 的引用。
4. 每个属性或构造方法参数值会被转换特定格式的形式，去匹配属性或构造方法参数的类型。默认情况下，Spring 可以转换给定的字符串格式的值到内建的类型，比如 int，long， String，boolean 等。

当容器被创建时，Spring 容器会来验证每个 bean 的配置，包括验证 bean 的引用属性是 否是一个合法的 bean。然而，bean 属性本身直到 bean 真正被创建出来后才被设置进去。 当容器被创建时，bean 的范围是单例时，会被设置成预实例（默认情况）来创建。范围在 4.5 节，“bean 的范围”部分来解释。否则，bean 就会当被请求的时候被创建。一个 bean 的创建潜在地会引起一系列 bean 被创建，因为 bean 的依赖和它依赖的依赖（等等）要不 创建和定义出来。

循环依赖 如果你使用主要的构造方法注入，就可能会引起不可解决的循环依赖情形。 比如，类 A 需要通过构造方法注入获得类 B 的实例，而类 B 也需要通过构造方法注入获得类 A 的实例。如果把类 A 和类 B 进行相互注入，Spring 的 IoC 容器会在运行时检 测到这是循环引用的情况，并且抛出 BeanCurrentlyInCreationException 异常。

一个可行的方案是编辑一些要配置的类的源码，通过 setter 方法而不是构造方法进行注入。而且，避免构造方法注入并仅仅使用 setter 方法注入。换句话说，尽管这不是推荐做法，你可以通过 setter 方法注入来配置循环依赖。

不像典型的用例（没有循环依赖），在 bean A 和 bean B 之间的循环依赖强制一个 bean 被注入到另一个中会先于被完全初始化自己（就是经典的鸡和蛋的问题）。

通常情况下你可以信任 Spring 做出正确的事情。它会在容器加载时来检测配置问题， 比如引用了一个不存在的 bean 和循环依赖问题。当 bean 被实际创建出来后，Spring 设置属 性和解决依赖会尽量地晚些。这就意味着 Spring 容器已经加载正确了但晚些时候可能会生 成异常，比如当你请求一个创建时发生问题的对象或者是它的依赖发送问题。例如，bean 因为丢失或非法属性而抛出异常。这种潜在的一些配置问题的可见度延迟也就是为什么 ApplicationContext 的实现类默认情况都是预实例的单例 bean。在前期的时间和内存 消耗中，在 bean 真正需要前来创建这些 bean，当 ApplicationContext 被创建的时候， 你会发现这些配置问题，这还不晚。你也可以覆盖这个默认的行为，那么单例 bean 就会延 迟加载，而不是预实例的了。

如果没有循环依赖的存在，当一个或多个协作的 bean 被注入到一个独立的 bean 时， 每个协作的 bean 就会在被注入之前完全被配置好。这就意味着如果 bean A 对 bean B 有依 赖，那么 Spring 的 IoC 容器会完全配置 bean B 而优先于调用 bean A 中的 setter 方法。换句 话说，bean 被实例化了（而不是预实例的单例 bean），它的依赖才被注入，相关的生命周 期方法（比如配置初始化方法（4.6.1.1 节）或 InitializingBea n 的回调方法（4.6.1.1 节））才 被调用。

4.4.1.4 依赖注入示例

下面的示例使用了基于 XML 的配置元数据来进行基于 setter 方法的依赖注入。Spring XML 配置文件的一小部分来指定几个 bean 的定义：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- 使用嵌入<ref/>的元素来进行setter方法注入<ref/> -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>
    <!-- 使用整洁的'ref'属性来进行setter方法注入-->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {
    private AnotherBean beanOne; private YetAnotherBean beanTwo;
 private int i;
    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }
    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }
    public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

上面的示例中，setter 方法被声明为匹配 XML 文件中指定的属性。下面的示例使用基于 构造方法的依赖注入：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- 使用嵌入的<ref/>元素进行构造方法注入 -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>
    <!-- 使用整洁的'ref'属性来进行构造方法注入 -->
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg type="int" value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {
    private AnotherBean beanOne; private YetAnotherBean beanTwo;
 private int i;
    public ExampleBean(AnotherBean anotherBean, YetAnotherBean y
etAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }
}
```

在 bean 中指定的构造方法参数会被用于 ExampleBean 的构造方法参数。 现在考虑一下这个示例的变化情况，要代替使用构造方法，Spring 被告知调用 static

工厂方法并返回一个对象的实例：

```
    <bean id="exampleBean" class="examples.ExampleBean" factory-
method="createInstance">
    <constructor-arg ref="anotherExampleBean"/>
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
public class ExampleBean {
    // 私有的构造方法
    private ExampleBean(...) {
        ...
    }
    // 静态工厂方法; 这个方法的参数可以被认为是要返回bean的依赖，
    // 而不管那些参数是如何被使用的。
    public static ExampleBean createInstance (AnotherBean anothe
rBean,
        YetAnotherBean yetAnotherBean, int i) {
        ExampleBean eb = new ExampleBean (...);
        // 其它的操作...
        return eb;
    }
}
```

static 工厂方法的参数通过 `<constructor-arg/>` 元素来提供，这和构造方法已经被实际调用是完全一致的。由工厂方法返回的类的类型不需要和包含 static 工厂方法类 的类型相同，尽管在本例中是相同的。实例（非静态）工厂方法可以被用于本质上相同的方 式（除了 factory-bean 属性的使用，来代替 class 属性），所以这里不讨论它们的细节。

4.4.2 深入依赖和配置

正如之前章节中所提到的，你可以定义 bean 的属性和构造方法参数作为其它被管理 bean（协作者）的引用，或者作为内联值的定义。出于这个目的，Spring 的基于 XML 的配 置元数据支持使用 `<property/>` 和 `<constructor-arg/>` 元素的子元素类型。

4.4.2.1 直接值（原生类型，String，等）

元素的 value 属性指定了属性或构造方法参数，这是人们可以阅读的 字符串的表达。正如之前提到过的（6.4.2 节），JavaBean 的 PropertyEditor 可以用来转 换这些字符串值从 String 到属性或参数的真实类型。

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicData
Source" destroy-method="close">
    <!-- setDriverClassName(String)方法调用的结果 -->
    <property name="driverClassName" value="com.mysql.jdbc.Drive
r"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb
"/>
    <property name="username" value="root"/>
    <property name="password" value="masterkaoli"/>
</bean>
```

对于更简洁的 XML 配置，下面的示例使用了 p-命名空间（4.4.2.6 节）。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
    http://www.springframework.org/schema/beans/spring -beans-3.
0.xs d">
    <bean id="myDataSource" class="org.apache.commons.dbcp.Basic
DataSource" destroy-method="close" p:driverClassName="com.mysql.
jdbc.Driver" p:url="jdbc:mysql://localhost:3306/mydb" p:username
="root" p:password="masterkaoli"/>
</beans>
```

上面的 XML 非常简洁；然而，错误之处会在运行时被发现而不是设计的时候，除非在 你创建 bean 的时候，使用如 IntelliJ IDEA 或 SpringSource Tool Suite（STS，SpringSource 组织 开发的工具套件）这样的 IDE 来支持自动地属性补全。这样的 IDE 帮助是强烈建议使用的。

你也可以这样来配置 java.util.Properties 实例：

```
<bean id="mappings" class="org.springframework.beans.factory.con
fig.PropertyPlac
    eholderConfigurer">
    <!-- 作为java.util.Properties类型 -->
    <property name="properties">
        <value>
            jdbc.driver.className=com.mysql.jdbc.Driver jdbc.url
=jdbc:mysql://localhost:3306/mydb
        </value>
    </property>
</bean>
```

Spring 容器会使用 JavaBean 的 PropertyEditor 机制来转换 `<value/>` 元素中的文本 到 java.util.Properties 实例。这是一个很好的捷径，也是 Spring 团队少有喜欢使用 嵌套的 `<value/>` 元素而不是 value 属性方式的地方之一。

idref 元素

idref 元素是一种简单防错的形式来传递容器中另外一个 bean 的 id（字符串值而不是引用）到 `<constructor-arg/>` 或 `<property/>` 元素中。

```
<bean id="theTargetBean" class="..."/>
<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean" />
    </property>
</bean>
```

上面定义 bean 的代码片段是和下面的片段完全等价（在运行时）：

```
<bean id="theTargetBean" class="..." />
<bean id="client" class="...">
    <property name="targetName" value="theTargetBean" />
</bean>
```

第一种形式比第二种形式可取，因为使用 idref 标签允许容器在部署时来验证被引用 的，命名的 bean 是否真的存在。第二种形式下，没有对传递给 client bea n 的 targetName 属性执行验证。错误仅仅是在 client bean 被真正实例化的时候才会被发现（和很多可能 致命的结果）。如果 client bean 是 prototype（4.5 节）类型的 bean，这个错误和结果异常 可能仅仅在容器被部署很长一段时间后才会被发现。

此外，如果被引用的 bean 在同一个 XML 单元中，bean 的名称就是 bean 的 id，那么你 还可以使用 local 属性，这允许 XML 解析器本身在 XML 文档解析的时候，及早地来验证 bean 的 id。

```
<property name="targetName">
    <!-- id为'theTargetBean'的bean必须存在;否则就会抛出异常 -->
    <idref local="theTargetBean"/>
</property>
```

`<idref/>` 元素带来的一个相同之处（最少是 Spring 2.0 版本以前的）是值在 ProxyFactoryBean 定义的 AOP 拦截器的配置中。当你指定拦截器名称来防止拼错拦截 器的 id 时，可以使用 `<idref/>` 元素。

4.4.2.2 引用其它 bean（协作者）

ref 元素是 `<constructor-arg/>` 或 `<property/>` 定义元素中最后的一个。在这里 你可以为 bean 设置指定属性的值，来引用被容器管理的另外一个 bean（协作者）。被引用 的 bean 就是要设置属性的这个 bean 的一个依赖，并且是初始化的作为属性设置之前的点 播。（如果协作者是一个单例的 bean，它可能已经被容器初始化过了。）所有引用最终都会 被引用到一个对象中。范围和验证基于通过 bean，local 或 parent 属性指定 id/name 的其它对象。

通过 `<ref/>` 标签的 bean 属性来指定目标 bean 是最常用的方式，并允许创建引用到相 同容器或父容器的任意 bean 中，而不管它们是不是在同一个 XML 文件中。bean 属性的值 可能和目标 bean 的 id 属性相同，或者是目标 bean 的 name 属性值之一。

```
<ref bean="someBean"/>
```

通过 local 属性来指定目标 bean 是利用了 XML 解析器的能力，来验证 XML 相同文件 内的 id 引用。local 属性的值必须和目标 bean 的 id 属性一致。如果没有在相同的文件内 发现匹配的元素，那么 XML 解析器会报告问题。因此，如果目标 bean 在同一个 XML 文件 中的话，使用 local 这种形式是最佳选择（为了更早地知道错误）。

```
<ref local="someBean"/>
```

通过 parent 属性来指定目标 bea n 会为 bea n 创建引用，它是在当前容器的父容器中的。parent 属性的值可能和目标 bean 的 id 属性相同，或者是目标 bean 的 name 属性值 之一，而且目标 bean 必须在当前容器的父容器中。当有一个容器继承关系，可以使用这个 bean 的引用，或在你想使用和父 bean 有相同名称的代理包装一个父容器中已有 bean 时。

```
<!-- 在父上下文中-->
<bean id="accountService" class="com.foo.SimpleAccountService">
    <!-- 在这里插入所需要的依赖 -->
</bean>
<!-- 在子（后继）上下文中 -->
<bean id="accountService" <-- bean名称和父bean相同 -->
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/>
        <!-- 注意我们如何引用父bean -->
    </property>
    <!-- 在这里插入其它配置和所需的依赖 -->
</bean>
```

4.4.2.3 内部 bean

在 `<property/>` 或 `<constructor-arg/>` 元素内部的 `<bean/>` 元素的定义被称为内部 bean。

```
<bean id="outer" class="...">
    <!-- 为了替代为目标bean使用引用，简单内联定义目标bean -->
    <property name="target">
    <bean class="com.example.Person"> <!-- 这就是内部bean -->
        <property name="name" value="Fiona Apple"/>
        <property name="age" value="25"/>
    </bean>
    </property>
</bean>
```

内部 bean 的定义不需要定义 id 或 name；容器忽略这些值。也会忽略 scope 标识。内部 bean 通常是匿名的，而且范围通常是 prototype（4.5.2 节）的。注入内部 bean 到协作 bean 是不可能的，只能到包围它的 bean 中。

4.4.2.4 集合

在 `<list/>` ， `<set/>` ， `<map/>` 和 `<props/>` 元素中，你可以分别设置 Java 的 Collection 类型 List，Set，Map 和 Properties 的属性和参数。

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call
-->
    <property name="adminEmails">
        <props>
            <prop
            key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</pro
p>
        </props>
    </property>
    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value
>
            <ref bean="myDataSource" />
        </list>
    </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key ="a ref" value-ref="myDataSource"/>
        </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

Map 类型的 key 或 value 的值，或 set 的值，也可以是下列元素之一：

- bean
- ref
- idref
- list
- set
- map
- props
- value
- null

集合合并

在 Spring 2.0 中，容器支持集合的合并。应用开发人员可以定义父样式的 `<list/>`，`<map/>`，`<set/>` 或 `<props/>` 元素，子样式的 `<list/>`，`<map/>`，`<set/>` 或 `<props/>` 元素继承或重写来自父集合的值。也就是说，子集合的值是合并元素父与子集合中的元素，和子集合元素覆盖父集合中指定值的结果。

本节关于导论父子 bean 合并的机制。如果读者不熟悉父和子 bean 的定义，可能要先去 阅读一下相关章节（4.7 节）。

下面的示例说明了集合合并：

```
<beans>
    <bean id="parent" abstract="true" class="example.ComplexObje
ct">
        <property name="adminEmails">
            <props>
                <prop
                key="administrator">administrator@example.com</p
rop>
                <prop key="support">support@example.com</prop>
            </props>
        </property>
    </bean>
    <bean id="child" parent="parent">
        <property name="adminEmails">
            <!-- 合并在*子*集合定义中来指定 definition -->
            <props merge="true">
                <prop key="sales">sales@example.com</prop>
                <prop key="support">support@example.co.uk</prop>
            </props>
        </property>
    </bean>
<beans>
```

注意在 child bean 中的 adminEmails 属性下的 `<props/>` 元素中的 merge=true 属性的使用。当 child bean 被容器处理并实例化时，结果实例中有一个 adminEmails Properties 的集合，包含了合并子 bean 的 adminEmails 集合和父 bean 的 adminEmails 集合。

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

子 bean 的 Properties 集合的值设置继承了从父 `<props/>` 元素而来的所有属性，子 bean 中 support 的值覆盖了父 bean 中的值。

这种合并行为的应用和 `<list/>`，`<map/>` 和 `<set/>` 集合类型很相似。在 `<list/>` 元素特 定的情形下，和 List 集合类型相关的语义，也就是说，ordered 集合值的概念，是要维 护的；父值优先于所有子 list 的值。在 Map，Set

和 Properties 集合类型的情况下，没有顺序的存在。因此对于容器内部使用的，和 Map，Set 和 Properties 实现类型相关的 集合类型没有排序语义的作用。

集合合并的限制

不能合并不同类型的集合（比如 Map 和 List），如果你要尝试这么去做，那么就会抛 出 Exception。merge 属性必须在低级的，继承的，子 bean 中来指定；在父集合中指定 merge 属性是冗余的，也不会看到想要的合并结果。合并特性仅在 Spring 2.0 和更高版本中 可用。

强类型集合（Java 5 以上版本）

在 Java 5 或更高版本中，你可以使用强类型集合（使用泛型）。也就是说，可以声明一 个 Collection 类型，它可以仅仅包含 String 元素（作为示例）。如果你使用 Spring 来对强类型的 Collection 依赖注入到 bean 中，你可以利用 Spring 的类型转换来支持这样 强类型 Collection 实例的元素，在被加到 Collection 之前，可以转换成合适的类型。

```
public class Foo {
    private Map<String, Float> accounts;
    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
<beans>
    <bean id="foo" class="x.y.Foo">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </bean>
</beans>
```

当 foo bean 的 accounts 属性准备好注入时，关于强类型元素 `Map<String, Float>` 类型的泛型信息就通过反射机制准备好了。因此 Spring 的类型转换工具识别到各种元素的 值作为 Float 类型，字符串值 9.99，2.75 和 3.99 被转换成实际的 Float 类型。

4.4.2.5 null 和空字符串

Spring 将属性的空参数当作空 String。下面基于 XML 的配置元数据片段设置了电子邮 件属性为空 String 值（""）。

```
<bean class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

上面的例子和下面的 Java 代码是一样的：exampleBean.setEmail("")。`<null/>` 元素控制 null 值。比如：

```
<bean class="ExampleBean">
    <property name="email"><null/></property>
</bean>
```

上面的配置和下面的 Java 代码一致：exampleBean.setEmail(null)。

4.4.2.6 使用 p-命名空间的 XML 快捷方式

p-命名空间可以使你使用 bea n 的元素属性，而不是内嵌的 `<property/>` 元素，来描述属 性的值和/或协作的 bean。

Spring 2.0 和后续版本支持使用命名空间（附录 C）的可扩展的配置格式，这是基于 XML的 Schema 定义的。本章中讨论的 bea n 的配置格式是定义在 XML 的 Schema 下的。然而，p-命名空间则不是定义在 XSD 文件下的，仅仅存在于 Spring 的核心中。 下面的示例展示了两个 XML 片段，解析得到相同的结果：第一个使用了标准的 XML 格式，第二个使用了 p-命名空间。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/be
ans
    http://www.springframework.org/schema/beans/spring -beans-3.
0.xsd">
    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value=["foo@bar.co](mailto:foo@ba
r.com)m"/>
    </bean>
    <bean name="p-namespace" class="com.example.ExampleBean" p:e
mail=["foo@bar.co](mailto:foo@bar.com)m"/>
</beans>
```

示例展示了 p-命名空间下的属性，在 bean 的定义中称为 email。这就告诉 Spring 来包 含属性声明。正如前面提到的，p-命名空间没有 schema 定义，所以你可以设置属性的名称 和 bean 中属性的名称一样。

下面的示例包含了两个 bean 的定义，两者都有对其它 bean 的引用：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema -instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation"http://www.springframework.org/schema/bea
ns
    http://www.springframework.org/schema/beans/spring -beans-3.
0.xsd">
    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>
    <bean name="john-modern" class="com.example.Person" p:name="
John Doe" p:spouse-ref="jane"/>
    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

正如你所看到的一样，这个例子包含着不仅仅使用了 p-命名空间的属性值，也使用了 特殊格式来声明属性的引用。在第一个 bean 中 使用了 `<property name="spouse" ref="jane"/>` 来创建 john bean 对 jane bean 的引用关系，第二个 bean 中，使用了 p:spouse-ref="jane"作为属性来做相同的事情。本例中，spouse 是属性名，而-ref 部分表明了这不是一个直接值而是一个对其它 bean 的引用。

> 注意
>
> P-命名空间没有标准 XML 格式那么灵活。比如，声明属性引用的格式和以 Ref 结尾的 属性相冲突，而标准的 XML 格式则不会。我们建议谨慎选择所用的方法并和开发团队成员 充分地交流，避免产生同时使用这三种方法的 XML 文档。

4.4.2.7 使用 c-命名空间的 XML 快捷方式

和 4.4.2.6 节，“使用 p-命名空间的 XML 快捷方式”相似，c-命名空间是在 Spring 3.1 中 被引入的，允许使用内联属性来配置构造方法参数而不是使用 constructor-arg 元素。

我们来看 4.4.1.1 节，“基于构造方法的依赖注入”中的示例，现在使用 c 命名空间：

```
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:p="http:
//www.springframework.org/schema/c" xsi:schemaLocation="http://w
ww.springframework.org/schema/beans http://www.springframework.o
rg/schema/beans/spring-beans.xsd">
    <bean id="bar" class="x.y.Bar"/>
    <bean id="baz" class="x.y.Baz"/>
    <-- '传统的'声明方式 -->
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
        <constructor-arg value=["foo@bar.co](mailto:foo@bar.com)
m"/>
    </bean>
    <-- 'c-命名空间'声明方式 -->
    <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="ba
z" c:email=["foo@bar.co](mailto:foo@bar.com)m">
</beans>
```

c:命名空间使用了和 p:（以-ref 结尾的 bean 引用）相同的转换机制通过它们的名称 来设置构造方法参数。这样一来，即便没有在 XSD schema（但是存在于 Spring 核心的内部） 中定义，它还是需要声明出来。

在极少数的情况下，构造方法参数名称是不可用的（通常如果字节码在编译时没有调试 信息），我们可以使用参数索引：

```
<-- 'c-命名空间'索引声明 -->
<bean id="foo" class="x.y.Foo" c:_0-ref="bar" c:_1-ref="baz">
```

> 注意
>
> 因为 XML 的语法，索引符号需要在其头部使用_作为 XML 属性名称，因为 XML 中属性 是不能以数字开头的（尽管一些 IDE 允许这样做）。

在实际运用中，构造方法解析机制（4.4.1.1 节）在匹配参数时是非常有效率的，所以除 非真有需要，我们建议在配置中使用名称符号。

4.4.2.8 复合属性名称

在设置 bean 的属性时，只要路径中的所有组件，除了最后一个属性名称是非 null 的， 可以使用复合或者嵌套的属性名称。参考下面的 bean 定义。

```
<bean id="foo" class="foo.Bar">
    <property name="fred.bob.sammy" value="123" />
</bean>
```

foo bean 有一个 fred 属性，它还有一个 bob 属性，而它仍有一个 sammy 属性，而最 终的 sammy 属性被设置成数值 123。为了让这样的配置可用，foo 的属性 fred，fred 的 属性 bob 在 bean 被构造好后必须不能为 null，否则会抛出 NullPointerException 异常。

4.4.3 使用 depends-on

如果一个 bean 是另外一个 bean 的依赖，通常表明了它会被设置成另外一个的属性。 典型的情况是在基于 XML 的配置元数据中使用 `<ref/>` （4.4.2.2 节）元素来完成。然而，有 时在两个 bean 之间的依赖并不是直接的；比如，类中的静态初始化器需要触发，就像数据 库驱动程序的注册。depends-on 属性可以明确地强制一个或多个 bean 在使用这个元素的 bean 被初始化之前被初始化。下面的示例使用了 depends-on 属性来表示一个独立 bean 的依赖：

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

为了表示对多个 bean 的依赖，提供一个 bean 名称的列表作为 depends-on 属性的值 即可，要用逗号，空白或分号分隔开，它们都是有效的分隔符：

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accou
ntDao">
    <property name="manager" ref="manager" />
</bean>
<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

> 注意
>
> 在 bean 定义中的 depends-on 属性可以指定初始化时的依赖，也可以是仅仅是单例（4.5.1 节）bean，对应销毁时的依赖。和给定 bean 定义了 depends-on 关系的依赖 bean 首先被销毁，先于给定的 bean 本身。因此 depends-on 也能控制关闭顺序。

4.4.4 延迟初始化 bean

默认情况下，ApplicationContext 的实现类积极地创建并配置所有单例的 bean （4.5.1 节），作为初始化过程的一部分。通常来说，这种预实例化是非常可取的，因为在配 置或周边环境中的错误可以直接被发现，而不是几小时或几天后去发现。当这种行为不可用时，你可以阻止单例 bean 的预实例化，在 bean 定义中使用延迟初始化来标记一下就可以 了。延迟初始化 bean 告诉 IoC 容器在该 bean 第一次被请求时再来实例化，而不是在启动时 实例化。

在 XML 中，这个行为可以通过 `<bean/>` 元素的 lazy-init 属性来控制；比如：

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init=
"true"/>
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

当前者配置被 ApplicationContext 处理时，命名为 lazy 的 bean 并不会在

ApplicationContext 启动时被预实例化，而 not.lazy bean 会被先预实例化。 而到延迟初始化 bean 是一个单例 bean 的依赖时，且这个单例 bean 不是延迟初始化的，那么 ApplicationContext 也会在启动时来创建延迟初始化的 bean，因为它必须满足单 例 bean 的依赖。延迟初始化的 bean 被注入到单例 bean 中的时候它就不是延迟初始化的了。

你也可以在容器级别来控制延 迟初始化，在 `<beans/>` 元素上使用default-lazy-init 属性；比如：

```
<beans default-lazy-init="true">
    <!-- 那么就没有bean是预实例化的了... -->
</beans>
```

4.4.5 自动装配协作者

Spring 容器可以自动装配协作 bean 之间的关系。你可以允许 Spring 自动为你的 bean 来 解析协作者（其它 bean），通过检查 ApplicationContext 的内容即可。自动装配有下 列优势：

自动装配可以显著减少指定属性或构造方法参数的需要。（如 bean 模板的机制，这将在 本章的 4.7 节来讨论，在这方面是有价值的。）

自动装配可以更新配置对象的演变。比如，如果你需要给一个类添加依赖，那个依赖可 以自动被填入而不需要修改配置。因此，自动装配在开发期间是非常有用的，当代码库 变得更稳定时切换到明确的装配也没有否定的选择。

当使用基于 XML 的配置元数据（4.4.1 节）时，你可以为 bean 的定义指定自动装配模 式，在 `<bean/>` 元素上设置 autowire 属性即可。自动装配功能有五种模式。你可以为每个 bean 来指定自动装配，因此可以选择为哪一个来指定自动装配。

表 4.2 自动装配模式

模式	解释
no	（默认情况）没有自动装配。Bean 的引用必须通过 ref 元素来定义。对于大型的部署，修改默认设置是不推荐的，因为明确地指定协作者会给予更多的控制和清晰。某种程度上来说，它勾勒出了系统的结构。
byName	通过属性名称来自动装配。Spring 以相同名称来查找需要被自动装配的 bean。比如，如果 bean 被设置成由名称来自动装配，并含有一个 master 属性（也 就说，有 setMaster(..)方法），Spring 会查找名为 master 的 bean 定义，并且用它来设置属性。
byType	如果 bean 的属性类型在容器中存在的话，就允许属性被自动装配。如果存在多于一个，就会抛出致命的异常，这就说明了对那个 bean 不能使用 byType 进行自动装配。如果没有匹配的 bean 存在，就不会有任何效果；属性不会 被设置。
constructor	和 byType 类似，但是是应用于构造方法参数的。如果在容器中没有确定的构造方法参数类型的 bean 的存在，就会发生致命的错误。

使用 byType 或 constructor 自动装配模式，你可以装配数组和集合类型。这种情况下所有在容器内匹配期望类型的自动装配候选者会被用来满足依赖。如果期望的键类型是String，你可以自动装配强类型的 Map。自动装配的 Map 值会包括匹配期望类型的实例， 而且 Map 的键会包含对应 bean 的名称。

你可以联合自动装配行为和依赖检查，这会在自动装配完成之后执行。

4.4.5.1 自动装配的限制和缺点

当在项目中一直使用时，自动装配是非常不错的。如果自动装配通常是不使用的，它就 可能会迷惑开发人员来使用它去装配仅仅一两个 bean。

考虑一下自动装配的限制和缺点：

- 在 property 和 constructor-arg 中设置的明确依赖通常覆盖自动装配。不能自动 装配所谓的简单属性，比如原生类型，String 和 Class（还有这样简单属性的数字）。这 是由于设计的限制。
- 自动装配没有明确装配那么精确，尽管，在上面的表格中已经说明了，Spring 很小心地 避免在有歧义的情况下去猜测，但也可能会有意想不到的结果，在 Spring 管理的对象 中间的关系可能就不会再清晰地说明了。
- 装配信息可能对从 Spring 容器中来生成文档的工具来说不可用。
- 在容器中的多个 bean 定义可能通过 setter 方法或构造方法参数匹配指定的类型进行自 动装配。对于数组，集合或 Map，这不一定是一个问题。而对期望简单值的依赖，这 种歧义不能随意解决。如果没有唯一的 bean 可用，就会抛出异常。 在后面一种情况中，你有几种选择：

- 放弃自动装配而使用明确的装配。
- 避免设置它的 autowire-candidate 属性为 false 来自动装配 bean，这会在下一节中来解释。
- 设置 `<bean/>` 元素的 primary 属性为 true 来指定单独的 bean 作为主要的候选者。
- 如果你使用 Java 5 或更高版本，使用基于注解的配置实现更细粒度的控制，这会在 4.9 节，“基于注解的容器配置”中来解释。

4.4.5.2 从自动装配中排除 bean

在每个 bean 的基础上，你可以从自动装配中来排除 bean。在 Spring 的 XML 格式配置 中，设置 `<bean/>` 元素的 autowire-candidate 属性为 false；容器会把指定的 bean 对自动装配不可用（包含注解风格的配置，比如@Autowired（4.9.2 节））。

你也可以基于 bean 名称的模式匹配来限制自动装配候选者。顶级的 `<beans/>` 元素中的 default-autowire-candidates 属性接受一个或多个模式。比如，为了限制自动装 配候选者到任意名称以 Repository 结尾 bean 的状态，提供*Repository 值。要提供多个模式，把它们定义在以逗号分隔的列表中。Bean 定义中 autowire-candidate 属性的 true 或 false 明确的值通常是优先的，而且对于这些 bean 来说，模式匹配规则是不适用的。 这些技术对那些永远不想通过自动装配被注入到其它 bea n 中的 bea n 来说是很有用的。

这并不意味着未包含的 bean 不能使用自动装配来配置。相反，bean 本身不是自动装配其它 bean 的候选者。

4.4.6 方法注入

在很多应用场景中，很多容器中的 bean 是单例（4.5.1 节）的。当一个单例的 bean 需 要和其它单例的 bean 协作时，或者一个非单例的 bean 需要和其它非单例的 bean 协作时， 典型地做法是通过定义一个 bean 作为另外一个的属性来控制依赖。当 bean 的生命周期不 同时问题就产生了。假设单例 bean A 需要使用非单例（prototype，原型）bean B，或许在 A 的每个方法调用上。容器仅仅创建单例 bean A 一次，因此仅仅有一次机会去设置属性。容 器不能每次为 bean A 提供所需的 bean B 新的实例。

解决方法就是放弃一些控制反转。你可通过实现 ApplicationContextAware 接口 以让 bean A 去感知容器（4.6.2 节），而且在每次 bean A 需要 bean B 的实例时，可以让容器 调用 getBean(“B”)（4.2.3 节）去获取（一个新的）bean B。下面的代码就是这种方式的例子：

```
// 使用了有状态的命令风格的类来执行一些处理
package fiona.apple;
// 导入Spring的API
import org.springframework.beans.BeansException;
import org.springframework.context.Applicationcontext;
import org.springframework.context.ApplicationContextAware;
public class CommandManager implements ApplicationContextAware {
    private ApplicationContext applicationContext;
    public Object process(Map commandState) {
        // 抓取相应命令的新实例
        Command command = createCommand();
        // 在命令实例上（希望标记新的）设置状态
        command.setState(commandState);
        return command.execute();
    }
    protected Command createCommand() {
        // 注意Spring API的依赖！
        return this.applicationContext.getBean("command", Comman
d.class);
    }
    public void setApplicationContext(ApplicationContext applica
tionContext)
        throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

前面的做法是不可取的，因为这些业务代码感知并耦合到了 Spring 框架中。方法注入， 这种 Spring IoC 容器中的有点先进的功能，允许以一种干净的方式来处理这个用例。

你可以在博客文章中来阅读更多关于方法注入的动机。

4.4.6.1 查找方法注入

查找方法注入是容器在其管理的 bean 上来覆盖方法的能力，来返回容器中其它命名 bean 的查找结果。查找，典型的情况是涉及原型 bean，这是在之前章节中描述的情景。Spring Framework 从 CGLIB 类库中，通过使用字节码生成机制实现了这种方法注入，来动态地生成 覆盖方法的子类。

> ⓘ
>
> 注意
>
> 要让这些动态子类起作用，在类路径下必须有 CGLIB 的 jar 文件。这些 Spring 容器中的 子类不能是 final 类型的，要被覆盖的方法也不能是 final 类型的。而且，测试有 abstract 方法的类要求你自己去编写类的子类并提供 abstract 方法的实现。最后，是 方法注入目标的对象还不能被序列化。

看一下之前代码片段中的 CommandManager 类，你会看到 Spring 容器会动态地覆盖 createCommand()方法的实现。CommandManager 类不会有任何 Spring 的依赖，在重新设计 的例子中你会看到：

```
package fiona.apple;
// 没有Spring API的导入!
public abstract class CommandManager {
    public Object process(Object commandState) {
        // 抓取一个心得合适的Command接口的实例
        Command command = createCommand();
        //在命令实例上（希望标记新的）设置状态
        command.setState(commandState);
        return command.execute();
    }
    // 好的... 但是这个方法的实现在哪儿?
    protected abstract Command createCommand();
}
```

在客户端类中包含要被注入（本例中的 CommandManager）的方法，要被注入的方法 需要从下面的示例中编写一个签名：

```
<public|protected> [abstract] <return-type> theMethodName(no-arg
uments);
```

如果方法是 abstract 的，动态生成的子类就实现这个方法。否则，动态生成的子类覆盖 定义在源类中的确定方法。比如：

```
<!-- 部署为原型的（非单例的）有状态的bean -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="proto
type">
    <!-- 注入所需要的依赖 -->
</bean>
<!-- commandProcessor使用statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
    <lookup-method name="createCommand" bean="command"/>
</bean>
```

当需要 command bean 的新的实例时，标识为 commandManager 的 bean 调用它自己的 方法 createCommand()。部署 command bean 为原型的可必须要小心，那确实就是需要 的才行。如果被部署为单例的（4.5.1 节），那么每次会返回相同的 command bean 的实例。

提示

感兴趣的读者可能会发现要使用 ServiceLocatorFactoryBean（在 org.springframework.beans.factory.config 包下）。在 ServiceLocatorFactoryBean 中的使用的方法和其它工具类是相似的，ObjectFactoryCreatingFactoryBean，但是它允许你去指定你自己的查找接口而不是 Spring 特定的查找接口。对这些类查询一下 JavaDoc 文档，还有博客文章来获取 ServiceLocatorFactoryBean 的额外的信息。

4.4.6.2 任意方法的替代

在方法注入中，用处不如查找方法注入大的一种形式，就是使用另外一个方法实现来替 换被容器管理的 bean 中的任意方法的能力。用户可以安全地跳过本节的其它部分，直到你 真正需要这些功能的时候来回过头来看。

使用基于 XML 的配置元数据，对于部署的 bean，你可以使用 replaced-method 元素来使用另外一个方法替换已有的方法实现。看看下面这个类，有一个我们想要去覆盖的 computeValue 方法：

```
public class MyValueCalculator {
    public String computeValue(String input) {
        // 真正的代码...
    }
    // 其它方法...
}
```

实现了 org.springframework.beans.factory.support.MethodReplacer 接口的类提供新的方法定义。

```
/** 也就是要在MyValueCalculator中实现用来覆盖已有的方法
computeValue(String)
*/
public class ReplacementComputeValue implements MethodReplacer {
    public Object reimplement(Object o, Method m, Object[] args)
    throws Throwable {
        // 获取输入值，并处理它，返回一个计算的结果
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

部署源类的 bean 定义和指定的方法覆盖可以是这样的：

```
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">
<!-- 任意方法替换 -->
    <replaced-method name="computeValue" replacer="replacementCo
mputeValue">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>
<bean id="replacementComputeValue" class="a.b.c.ReplacementCompu
teValue"/>
```

你可以在 `<replaced-method/>` 元素中使用一个或多个包含的 `<arg-type/>` 元素来 指示被覆盖方法的方法签名。如果方法是被重载的并在类中有很多种形式的存在，那么参数 的签名是必须的。为了简便，字符串类型的参数可能是类型全名的一个子串。比如，下面所 有的都匹配 java.lang.String：

```
java.lang.String
String
Str
```

因为用参数的数量用来区分每一个可能的选择通常是足够的，这种快捷方式可以节省大 量的输入，允许你去输入匹配参数类型的最短的字符串。

4.5 Bean 的范围

当创建 bean 时，也就创建了对通过 bean 定义创建的类真正实例的配方。bean 定义的 配方这个概念是很重要的，因为它就意味着，你可以通过配方来创建一个类的多个对象实例。

你不仅可以控制从特定 bean 定义创建出的对象的各个依赖和要配置到对象中的值，也 可以控制对象的范围。这个方法非常强大并且很灵活，你可以选择通过配置创建的对象的范 围，而不必去处理 Java 类级别对象的范围。Bean 可以被定义部署成一种范围：开箱，Spring

Framework 支持五种范围，里面的三种仅仅通过感知 web 的 ApplicationContext 可用。

下列的范围支持开箱。你也可以创建自定义范围（4.5.5 节）。

表 4.3 Bean 的范围

范围	描述
单例（4.5.1 节）	（默认的）为每一个 Spring 的 IoC 容器定义一个 bean 为独立对象的实例。
原型（4.5.2 节）	定义可以有任意个对象实例的 bean。
请求（4.5.4.2 节）	定义生命周期为一个独立 HTTP 请求的 bean；也就是说，每一个 HTTP 请求有一个 bean 的独立的实例而不是一个独立的 bean。仅仅在可感知 Web 的 Spring ApplicationContext 中可用。
会话（4.5.4.3 节）	定义一个生命周期为一个 HTTP Session 的独立的 bean。仅仅在可感知 Web 的 Spring ApplicationContext 中可用。
全局会话（4.5.4.4 节）	定义一个生命周期为一个全局的 HTTP Session 的独立的 bean。典型地是仅仅 使用 portlet 上 下文 时可 用。 仅仅 在可 感知 Web 的 Spring ApplicationContext 中可用。

线程范围的 bean

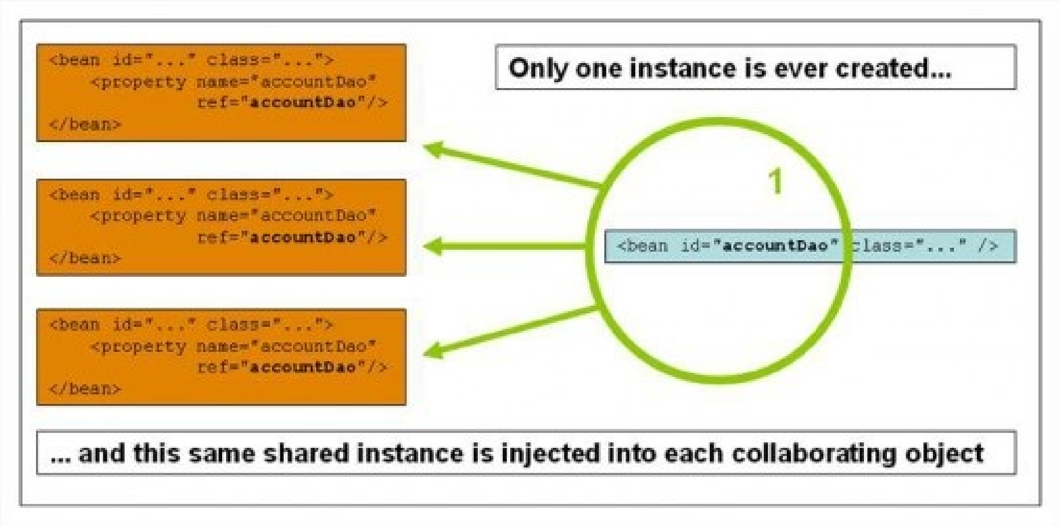
在 Spring 3.0 当中，线程范围是可用的，但是它默认是不注册的。要获取更多信息，请 参考 SimpleThreadSc ope 的 JavaDoc 文档。对于如何注册这个范围或其它自定义范围的做法，

请参考 4.5.5.2 节，“使用自定义范围”。

4.5.1 单例范围

仅仅共享被管理的 bean 的一个实例，所有使用一个或多个 id 来匹配 bean 的结果对 bean 请求，由 Spring 容器返回指定 bean 的实例。

另外一种方式，当你定义一个范围是单例的 bean 后，Spring 的 IoC 容器通过 bean 的定 义创建了这个对象的一个实例。这个独立的实例存储在单例 bean 的缓存中，所有对这个命 名 bean 的后续的请求和引用都返回缓存的对象。



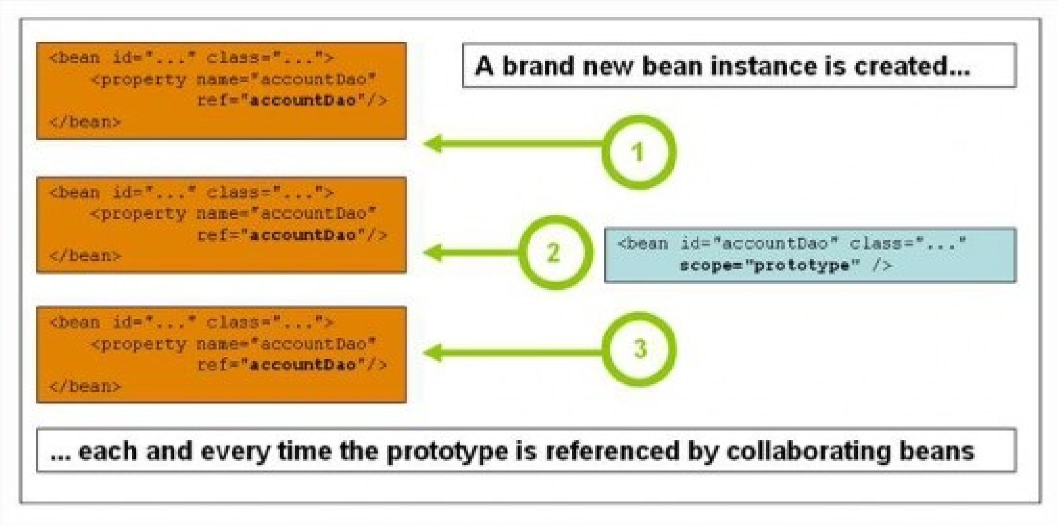
Spring 中对单例 bean 的概念和四人帮（Gang of Four，GoF）设计模式书中定义的单例 模式是不同的。GoF 的单例硬编码了对象的范围，也就是特定的类仅仅能有一个实例被每一 个 ClassLoader 来创建。Spring 中单例 bean 的范围，是对每一个容器和 bean 来说的。这 就意味着在独立的 Spring 容器中，对一个特定的类创建了一个 bean，那么 Spring 容器通过 bean 的定义仅仅创建这个类的一个实例。在 Spring 中，单例范围是默认的范围。要在 XML 中定义单例的 bean，可以按照如下示例来编写：

```
<bean id="accountService" class="com.foo.DefaultAccountService"/
>
<!-- 尽管是冗余的（单例范围是默认的），下面的bean定义和它是等价的 -->
<bean id="accountService" class="com.foo.DefaultAccountService"
scope="singleton"/>
```

4.5.2 原型范围

非单例，原型范围的 bean 就是当每次对指定 bean 进行请求时，一个新的 bean 的实例 就会创建。也就是说，bean 被注入到其它 bean 或是在容器中通过 getBean() 方法来请求时就 会创建新的 bean 实例。作为一项规则，对所有有状态的 bean 使用原型范围，而对于无状 态的 bean 使用单例范围。

下图讲述了 Spring 的原型范围。数据访问对象（DAO）通常不是配置成原型的，因为典型的 DAO 不会持有任何对话状态；仅仅是对作者简单对单例图的重用。



下面的示例在 XML 中定义了原型范围的 bean：

```
<!-- 使用spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService"
scope="prototype"/>
```

和其它范围相比，Spring 不管理原型 bean 的完整生命周期；容器实例化，配置并装配 原型对象，并把他给客户端，对于原型 bean 的实例来说，就没有进一步的跟踪记录了。因 此，尽管初始化生命周期回调方法可以对所有对象调用而不管范围，那么在原型范围中，配 置销毁生命周期回调是不能被调用的。客户端代码必须清理原型范围的对象并且释放原型 bean 持有的系统资源。要让 Spring 容器来释放原型范围 bean 持有的资源，可以使用自定义 bean 后处理器（4.8.1 节），它持有需要被清理的 bean 的引用。

在某些方面，关于原型范围 bean 的 Spring 容器的角色就是对 Java new 操作符的替代。 所有之后生命周期的管理必须由客户端来处理。（关于 Spring 容器中 bean 的生命周期的详细内容，可以参考 4.6.1 节，“生命周期回调”）

4.5.3 单例 bean 和原型 bean 依赖

当你使用单例范围的 bean 和其依赖是原型范围的 bean 时，要当心依赖实在实例化时 被解析的。因此，如果依赖注入了原型范围的 bean 到单例范围的 bean 中，新的原型 bean 就被实例化并且依赖注入到单例 bean 中。原型实例是唯一的实例并不断提供给单例范围的 bean。

假设你想单例范围的 bean 在运行时可以重复获得新的原型范围 bean 的实例。那么就 不能将原型范围的 bean 注入到单例 bea n 中，因为这个注入仅仅发生一次，就是在 Spring 容器实例化单例 bean 并解析和注入它的依赖时。如果你在运行时需要原型 bean 新的实例 而不是仅仅一次，请参考 4.4.6 节，“方法注入”。

4.5.4 请求，会话和全局会话范围

request，session 和 global session 范围仅仅当你使用感知 Web 的 Spring ApplicationContext 实现类可用（比如 XmlWebApplicationContext）。如果你在 常规的 Spring IoC 容器中使用如 ClassPathXmlApplicationContext 这些范围，那么就会因为一个未知的 bean 的范围而得到 IllegalStateException 异常。

4.5.4.1 初始化 Web 配置

要支持 request，session 和 global session 级别（Web 范围的 bean）范围的 bean，一些细微的初始化配置就需要在定义 bean 之前来做。（这些初始化的设置对标准的 范围，单例和原型，是不需要的。）

如果来完成这些初始化设置是基于你使用的特定 Servlet 容器的。

如果使用 Spring 的 Web MVC 来访问这些范围的 bean，实际上，是由 Spring 的 DispatcherServlet 或 DispatcherPortlet 来处理请求的，那就没有特殊的设置了：DispatcherServlet 和 DispatcherPortlet 已经可以访问所有相关的状态。

如果你使用支持 Servlet 2.4 规范以上的 Web 容器，在 Spring 的 DispatcherServle（比如，当 使 用 JSF 或 Struts 时 ） 之 外 来 处 理 请 求 ， 那 么 就 需 要 添 加 javax.servlet.ServletRequestListener 到 Web 应用的 web.xml 文件的声明中：

```
<web-app>
    ...
    <listener>
        <listener-class> org.springframework.web.context.request
.RequestContextLi
            stener
        </listener-class>
    </listener>
    ...
</web-app>
```

如果你使用老版的 Web 容器（ Serlvet 2.3 ）， 那 么 就 使 用 提 供 的 javax.servlet.Filter 实 现 类 。 如 果 你 想 在 Servlet 2.3 容 器 中 ， 在 Spring 的 DispatcherServlet 外部的请求中访问 Web 范围的 bean，下面的 XML 配置代码片段就必须包 含在 Web 应用程序的 web.xml 文件中。（过滤器映射基于所处的 Web 应用程序配置，所以 你必须以适当的形式来修改它。）

```
<web-app>
    ..
    <filter>
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestCont
 extFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

DispatcherServlet，RequestContextListener 和 RequestContextFilter 所有都可以做相同的事情，即绑定 HTTP 请求对象到服务于请求的 Thread 中。这使得 bean 在请求和会话范围内对后续的调用链可用。

4.5.4.2 请求范围

考虑一下下面的 bean 定义：

```
<bean id="loginAction" class="com.foo.LoginAction" scope="reques
t"/>
```

Spring 容器通过使用 LoginAction bean 的定义来为每个 HTTP 请求创建 LoginAction bean 的新的实例。也就是说，LoginAction bean 是在 HTTP 请求级别的范 围。你可以改变创建的实例内部的状态，怎么改都可以，因为从相同的 LoginAction bean 定义中创建的其它实例在状态上不会看到这些改变；它们对单独的请求都是独立的。当请求 完成处理，bean 在请求范围内被丢弃了。

4.5.4.3 会话范围

考虑一下下面的 bean 定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope
="session"/>
```

Spring 容器通过使用 UserPreferences bean 的定义来为每个 HTTP 会话的生命周期 创建 UserPreferences bean 的新的实例。换句话说，UserPreferences bean 在 HTTP

session 级别的范围内是有效的。正如 request-scoped 的 bean，你可以改变创建的实 例 内 部 的 状 态 ，怎 么 改 都 可以 ， 要 知 道其 它 HTTP session 实例也是使用从相同 UserPreferences bean 的定义中创建的实例，它们不会看到状态的改

变，因为它们对单 独的 HTTP session 都是独立的。当 HTTP session 最终被丢弃时，那么和这个 HTTP session 相关的 bean 也就被丢弃了。

4.5.4.4 全局会话范围

考虑一下下面的 bean 定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope
="globalSession"/>
```

global session 范围和标准的 HTTP Session 范围（上面 4.5.4.3 节讨论的）类似，但仅能应用于基于 portlet 上下文的 Web 应用程序。portlet 规范定义了全局的 Session 的 概念，该会话在所有 portlet 之间所共享来构建一个单独的 portlet Web 应用程序。在 global session 范围内定义的 bean 的范围（约束）就会在全局 portlet Session 的生命周期内。

如果你编写了一个标准的基于 Servlet 的 Web 应用，并且定义了一个或多个有 global session 范围的 bean，那么就会使用标准的 HTTP Session 范围，也不会抛出什么错误。

4.5.4.5 各种范围的 bean 作为依赖

Spring 的 IoC 容器不仅仅管理对象（bean）的实例，而且也装配协作者（依赖）。如果 你想注入（比如）一个 HTTP 请求范围的 bean 到另外一个 bean 中，那么就必须在有范围 bean 中注入一个 AOP 代理。也就是说，你需要注入一个代理对象来公开相同的公共接口作为有 范围的对象，这个对象也可以从相关的范围（比如，HTTP 请求范围）中获取真实的目标对 象并且委托方法调用到真正的对象上。

> ⓘ
>
> 注意
>
> 你不需要使用 `<aop:scoped-proxy/>` 来结合 singleton 和 prototype 范围的 bean 。如果你想为单例 bean 尝试创建有范围的代理，那么就会抛出 BeanCreationException 异常。

在下面的示例中，配置仅仅只有一行，但是这对理解“为什么”和其中的“怎么做”是至关重要的。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:aop="htt
p://www.springframework.org/schema/aop" xsi:schemaLocation="http
://www.springframework.org/schema/beans http://www.springframewo
rk.org/schema/beans/spring -beans-3.0.xsd http://www.springframe
work.org/schema/aop http://www.springframework.org/schema/aop/sp
ring -aop-3.0.xsd">
    <!-- HTTP 会话范围的bean作为代理 -->
    <bean id="userPreferences" class="com.foo.UserPreferences" s
cope="session">
        <!-- 这个元素影响周围bean的代理 -->
        <aop:scoped-proxy/>
    </bean>
    <!-- 单例范围的bean用上面的代理bean来注入 -->
    <bean id="userService" class="com.foo.SimpleUserService">
        <!-- 作为代理的userPreferences bean引用 -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

要创建这样的代理，就要插入一个子元素 `<aop:scoped-proxy/>` 到有范围的 bean 的 定义中。（如果要选择基于类的代理，那么需要在类路径下放置 CGLIB 的类库，参考下面的“选择创建代理类型”部分和“附录 C，基于 XML Schema 的配置”）为什么 request， session，globalSession 和自定义范围级别的 bean 需要 `<aop:scoped-proxy/>` 元 素？我们来看看下面的单例 bean 定义，并将它和你需要定义的上述范围来比较一下。（下面 的 userPreferences bean 的定义是不完整的。）

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope
="session"/>
<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

在上面 的例 子中 ，单 例 bean userManager 是用 HTTP session 范围 的 bean userPreferences 的引用注入的。这里突出的一点就是 userManager bean 是单例的； 那么对于每个容器来说，它只会被实例化一次，并且它的依赖（本例中只有一个依赖，就是 userPreferences bean）也只会被注入一次。这就意味着 userManager bean 只会对同 一个 userPreferences 对象进行操作，也就是说，是最初被注入的那一个对象。

这并不是你想要的行为，即注入生命周期范围短的 bea n 到生命周期范围长的 bea n 中， 比如注入 HTTP session 范围的协作 bean 作为依赖到单例 bean 中。相反，你需要一个单独的 userManager 对 象 ， 生 命 周 期 和 HTTP session 一 样 ， 你 需 要 一 个 userPreferences 对象来特定于 HTTP session。因此容器创建对象，公开相同的公共 接口，而 UserPreferences 类（理想情况下是 UserPreferences 实例的对象）从范围机制（HTTP 请求，session 等等）来获取真

正的 UserPreferences 对象。容器注入这 个代理对象到 userManager bean 中，这个 bean 不能感知 UserPreferences 的引用就 是代理。在这个示例中，当 UserManager 实例调用依赖注入的 UserPreferences 对象 的方法时，那么其实是在代理上调用这个方法。然后代理从（在本例中）HTTP session 中 获取真实的 UserPreferences 对象 ，并且委托方法调用到获取的真实 的 UserPreferences 对象中。

因此，当注入 request 范围，session 范围和 globalSession 范围的 bean 到协作 对象时，你需要下面的，正确的，完整的配置：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope
="session">
    <aop:scoped-proxy/>
</bean>
<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

选择创建代理类型

默认情况下，当 Spring 容器为被 `<aop:scoped-proxy/>` 元素标记的 bean 创建代理时， 基于 CGLIB 的类的代理就被创建了。这就是说在应用程序的类路径下需要有 CGLIB 的类库。 注意：CGLIB 代理仅仅拦截公有的方法调用！不会调用代理中的非公有方法；它们不会被委托到有范围的目标对象。

另外，你可以配置 Spring 容器有范围的 bean 创建标准的基于 JDK 接口的代理，通过指 定 `<aop:scoped-proxy/>` 元素的 proxy-target-class 属性值为 false。使用基于 JDK 接口的代理意味着你不需要在应用程序的类路径下添加额外的类库来使代理生效。然而， 它也意味着有范围 bean 的类必须实现至少一个接口，并且注入到有范围 bean 的所有协作 者必须通过它的一个接口来引用 bean。

```
<!—DefaultUserPreferences实现了UserPreferences接口 -->
<bean id="userPreferences" class="com.foo.DefaultUserPreferences
" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>
<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

关于选择基于类或基于接口的代理的更多详细信息，可以参考 8.6 节，“代理机制”。

4.5.5 自定义范围

在 Spring 2.0 当中，bean 的范围机制是可扩展的。你可以定义自己的范围，或者重新定 义已有 的范 围， 尽管 后者 被认 为是 不良 实践 而且 你不能 覆盖 内建 的 singleton 和 prototype 范围。

4.5.5.1 创建自定义范围

要 整 合 自 定 义 范 围 到 Spring 的 容 器 中 ， 那 么 需 要 实 现 org.springframework.beans.factory.config.Scope 接口，这会在本节中来介绍。 对于如何实现你自己的范围的想法，可以参考 Spring Framework 本身提供的 Scope 实现和 Scope 的 JavaDoc文档，这里会介绍你需要实现的方法的更多细节。

Scope 接口有四个方法来从作用域范围中获取对象，从作用域范围中移除对象还有允 许它们被销毁。

下面的方法从底层范围中返回对象。比如，会话范围的实现，返回会话范围的 bean（如 果它不存在，在绑定到会话后为将来的引用，这个方法返回 bean 的新的实例）。

```
Object get(String name, ObjectFactory objectFactory)
```

下面的方法从底层范围中移除对象。比如会话范围的实现，从底层的会话中移除会话范 围的 bean。对象应该被返回，但是如果特定名字的对象没有被发现的话可以返回 null。

```
Object remove(String name)
```

下面的方法注册当被销毁或者指定范围内的对象被销毁时，相关范围应用执行的回调函 数。参考 JavaDoc 文档或者 Spring 范围实现来获取更多关于销毁回调的信息。

```
void registerDestructionCallback(String name, Runnable destructi
onCallback)
```

下面的方法得到一个底层范围的会话标识符。这个标识符对每种范围都是不同的。对于 会话范围的实现，这个标识符可以是会话标识符。

```
String getConversationId()
```

4.5.5.2 使用自定义范围

在你编写并测试一个或多个自定义 Scope 实现之后，你需要让 Spring 容器感知你的新 的范围。下面的方法是用 Spring 容器注册新的 Scope 的核心方法。

```
void registerScope(String scopeName, Scope scope);
```

这个方法在 ConfigurableBeanFactory 接口中声明，也在 大多数的 ApplicationContext 实现类中可用，通过 BeanFactory 属性跟着 Spring。

registerScope(..)方法的第一个参数是和范围相关的唯一名称；在 Spring 容器本 身这样的名字就是 singleton 和 prototype。registerScope(..)方法的第二个参数 是你想注册并使用的自定义 Scope 实现的真实实例。

假设你编写了自定义的 Scope 实现，之后便如下注册了它。

> 注意
>
> 下面的示例使用了 SimpleThreadScope，这是包含在 Spring 当中的，但模式是没有 注册的。这些指令和你自己自定义的 Scope 的实现是相同的。

```
Scope threadScope = new SimpleThreadScope(); beanFactory.registe
rScope("thread", threadScope);
```

之后就可以为你的自定义 Scope 创建符合 Spring 规则的 bean 的定义：

```
<bean id="..." class="..." scope="thread">
```

使用自定义的 Scope 实现，并没有被限制于程序注册。你也可以进行声明式的 Scope注册，使用 CustomScopeConfigurer 类：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:aop="htt
p://www.springframework.org/schema/aop" xsi:schemaLocation="http
://www.springframework.org/schema/beans http://www.springframewo
rk.org/schema/beans/spring -beans-3.0.xsd
http://www.springframework.org/schema/aop http://www.springframe
work.org/schema/aop/spring -aop-3.0.xsd">
    <bean
        class="org.springframework.beans.factory.config.CustomSc
opeC onfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean class="org.springframework.context.sup
port.SimpleT hreadScope"/>
                </entry>
            </map>
        </property>
    </bean>
    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>
    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>
</beans>
```

> 注意
>
> 当在 FactoryBean 的实现中放置 `<aop:scoped-proxy/>` 时，那就是工厂 bean 本身被放 置到范围中，而不是从 getObject()方法返回的对象。

4.6 自定义 bean 的性质

4.6.1 生命周期回调

为了和容器管理 bean 的生命周期进行交互，你可以实现 Spring 的 InitializingBean 和 DisposableBean 接口。容器为前者调用 afterPropertiesSet()方法，为后者调 用 destroy()方法在 bean 的初始化和销毁阶段允许 bean 执行特定的动作。你也可以和容 器达到相同的整合，而不需要通过使用初始化方法和销毁方法的对象定义元数据来耦合你的类到 Spring 的接口中。

从内部来说，Spring Framework 使用了 BeanPostProcessor 的实现来处理任何回调 接口，它可以发现并调用合适的方法。如果你需要自定义特性或其它生命周期行为，Spring 不会提供开箱，你可以自行实现 BeanPostProcessor 接口。要获取更多信息，可以参考 4.8 节，“容器扩展点”。

除了初始化和销毁回调，Spring 管理的对象也会实现 Lifecycle 接口，所以那些对象 可以参与到启动和关闭过程，来作为容器自己的生命周期。

生命周期回调接口在本节中来说明。

4.6.1.1 初始化回调

org.springframework.beans.factory.InitializingBean 接口允许 bean 在 所有必须的属性被容器设置完成之后来执行初始化工作。InitializingBean 接口中仅仅 指定了一个方法：

```
void afterPropertiesSet() throws Exception;
```

推荐不要使用 InitializingBean 接口，因为它不必要地将代码耦合到 Spring 中。 另外，可以指定一个 POJO 初始化方法。在基于 XML 配置元数据的情形中，你可以使用 init-method 属性来指定签名为无返回值，无参数的方法的名称。比如，下面的定义：

```
<bean id="exampleInitBean"
    class="examples.ExampleBean"
    init-method="init"/>
public class ExampleBean {
    public void init() {
        // 做一些初始化工作
    }
}
```

这和下面的做法是完全相同的：

```
<bean id="exampleInitBean"
    class="examples.AnotherExampleBean"/>
public class AnotherExampleBean implements InitializingBean {
    public void afterPropertiesSet() {
        //做一些初始化工作
    }
}
```

但是没有耦合任何代码到 Spring 中。

4.6.1.2 销毁回调

实现 org.springframework.beans.factory.DisposableBean 接口，当容器 包含的 bean 被销毁时，允许它获得回调。DisposableBean 接口中仅仅有一个方法：

```
void destroy() throws Exception;
```

推荐不要使用 DisposableBean 接口，因为它不必要地将代码耦合到 Spring 中。另外， 可以指定一个由 bean 支持的通用的方法。使用基于 XML 配置元数据，你可以使用 `<bean/>` 元素中的 destroy-method 属性。比如，下面的定义：

```
<bean id="exampleInitBean"
    class="examples.ExampleBean"
    destroy-method="cleanup"/>
public class ExampleBean {
    public void cleanup() {
        // 做一些销毁工作 (比如释放连接池的连接)
    }
}
```

这和下面的做法是完全相同的：

```
<bean id="exampleInitBean"
    class="examples.AnotherExampleBean"/>
public class AnotherExampleBean implements DisposableBean {
    public void destroy() {
        // 做一些销毁工作 (比如释放连接池的连接)
    }
}
```

但是没有耦合代码到 Spring 中。

4.6.1.3 默认的初始化和销毁方法

当使用 Spring 指定的 InitializingBean 和 DisposableBean 回调接口来编写初 始化和销毁方法回调时，典型地做法就是编写名称为 init()，initialize()，dispose() 等方法.理想的情况，这样的生命周期回调方法的名称在一个项目中是标准化的，那么所有 的开发人员可以使用相同的方法名称来保证一致性。

你可以配置 Spring 容器在每个 bean 中来查看命名的初始化和销毁回调方法名称。这就 是说，作为应用程序开发人员，你可以编写应用类并且使用初始化回调方法 init()，而不 需要在每个 bean 的定义中来配置 init-method="init"属性。当 bean 被创建时（并按 照前面描述的标准的生命周期回调合约），Spring 的 IoC 容器调用这个方法。这个特性也对 初始化和销毁方法回调强制执行了一致的命名约定。

假设你的初始化回调方法命名为 init()，销毁回调方法命名为 destroy()。那么你 就可以按照如下示例来装配类：

```
public class DefaultBlogService implements BlogService {
    private BlogDao blogDao;
    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }
    // 这是（毋庸置疑的）初始化回调方法
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] prope
rty must be set.");
        }
    }
}
<beans default-init-method="init">
    <bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>
</beans>
```

顶层 `<beans/>` 元素的 default-init-method 属性的存在，就会让 Spring 的 IoC 容器意识到在 bean 中的一个叫做 init 的方法就是初始化回调方法。当一个 bean 被创建和装 配时，如果 bean 类有这样一个方法，那么它就会在合适的时间被调用。

相似地（也就是在 XML 中），你可以使用顶层元素 `<beans/>` 的 default-destroy-method 属性来配置销毁回调方法。

在已经存在的 bean 类中，有命名差异的回调方法，你可以指定（也就是在 XML 中） `<bean/>` 本身的 init-method 和 destroy-method 属性来覆盖默认的方法。

Spring 容器保证了在给 bean 提供所有的依赖后，配置的初始化回调方法会立即被调用。 因此初始化回调也被称为原始的 bean 引用，也就意味着 AOP 的拦截器等尚未应用到 bean 中。目标 bean 首先被完全创建，之后 AOP 代理（比方说）和它的拦截器链就会被应用上。

如果目标 bean 和代理分开来定义了，你的代码仍然可以绕开代理和原始目标 bean 来交互。 因此，引用拦截器到初始化方法中会是不一致的，因为这么来做了会和它的代理/拦截耦合 目标 bean 的合生命周期，当你的代码直接和原始目标 bean 交互时，会留下奇怪的语义。

4.6.1.4 组合生命周期机制

在 Spring 2.5 中，对控制 bean 的生命周期行为有三种选择：InitializingBean（4.6.1.1节）和 DisposableBean（4.6.1.2 节）回调接口；自定义 init()和 destroy()方法； @PostConstruct 和@PreDestroy 注解（4.9.6 节）。你可以组合这些机制来控制给定的 bean。

> 注意
>
> 如果对一个 bean 配置了多个生命周期机制，并且每种机制都用不同的方法名称来配置， 那么每个配置方法就会以下面列出的顺序来执行。然而，如果配置了相同的方法名称，比如说，init()对于初始化方法，多于一个生命周期机制，那么这个方法就执行一次，这在前 面章节解释过了。

相同的 bean 配置了不同初始化方法的多个生命周期机制，那么就按下面顺序调用：

- 使用@PostConstruct 注解的方法
- 由 InitializingBean 回调接口定义的 afterPropertiesSet()方法

自定义配置的 init()方法 销毁方法也是同样顺序调用：

- 使用@PreDestroy 注解的方法
- 由 DisposableBean 回调接口定义的 destroy ()方法
- 自定义配置的 destroy ()方法

4.6.1.5 启动和关闭回调

Lifecycle 接口定义了对于任意有它自己生命周期需求（比如，开始和结束一些后台进程）对象的必要方法：

```
public interface Lifecycle {
    void start();
    void stop();
    boolean isRunning();
}
```

任何 Spring 管理的对象可以实现这个接口。那么，当 ApplicationContext 本身启动和关 闭时，它会级联那些定义在上下文中，对所有生命周期实现的调用。它会委托给 LifecycleProcessor 来做：

```
public interface LifecycleProcessor extends Lifecycle {
    void onRefresh();
    void onClose();
}
```

要注意 LifecycleProcessor 本身是扩展了 Lifecycle 接口。而且它添加了两个 方法来对上下文的刷新和关闭做出反应。

启动和关闭调用的顺序是很重要的。如果一个“依赖”关系存在于两个任意对象间，那 么需要依赖的一方会在它的依赖启动之后启动，并且在它的依赖关闭之前停止。然而，有时 直接的依赖关系是未知的。你可能仅仅知道确定类型的对象应该优先于另外一种类型对象的启动。在那些情形下，SmartLifecycle 接口定义了另外一种选择，命名为 getPhase() 方法，在它的父接口 Phased 中来定义。

```
public interface Phased {
    int getPhase();
}
public interface SmartLifecycle extends Lifecycle, Phased {
    boolean isAutoStartup();
    void stop(Runnable callback);
}
```

当启动时，最低层的对象首先启动，当停止时，是相反的。因此，实现了 SmartLifecycle 接口，并且 getPhase()方法返回 Integer.MIN_VALUE 的对象就会在最 先开始和最后停止的中间。在范围的另外一端，Integer.MIN_VALUE 的层次值表示对象 应该最后启动并且最先停止（可能是因为它取决于其它进程的执行）。当考虑层次值时，知道任意“普通”对象并且没有实现 Lifecycle 接口的默认的层次是 0，这也是很重要的。 因此，任意负的层次值就表示对象应该在那些标准组件之前启动（在它们之后停止），对于 任意正的层次值，反之亦然。

正如你看到由 SmartLifecycle 定义的停止方法接收回调。任何实现类必须在实现类关闭进程完成之后，调用回调方法 run()。这使得必要时可以进行异步关闭，因为默认的 LifecycleProcessor 接口的实现，DefaultLifecycleProcessor,，会等待每个阶段一组对象来调用回调的超时值。每个阶段的默认超时是 30 秒。你可以在上下文中定义名 为“lifecycleProcessor”的 bean 来覆盖默认的生命周期进程实例。如果你仅仅想去修改超时 时间，那么定义下面的 bean 就足够了：

```
<bean id="lifecycleProcessor" class="org.springframework.context
.support.DefaultLifecycleProc essor">
    <!-- 毫秒数的超时时间 -->
    <property name="timeoutPerShutdownPhase" value="10000"/>
</bean>
```

正如所提到的，LifecycleProcessor 接口定义了刷新和关闭上下文的回调方法。后者将驱动关闭进程，就好像明确地调用了 stop()方法，但是当上下文关闭时它才会发生。另 一方面，‘刷新’回调使得 SmartLifecycle bean 的另外一个特性可用。当上下文刷新时（在所有对象都被实例化和初始化后），才会调用回调方法，在那时，默认的生命周期进程 会检查每个 SmartLifecycle 对象的 isAutoStartup()方法返回的布尔值。如果是“true”，那时对象将会被启动，而不是等待明确的上下文调用或它自己的 start()方法（不像上下文刷新，对标准的上下文实现来说，上下文启动不会自动发生）。“阶段”值和任意“依 赖”关系会按照上面所描述的相同方式来决定启动的顺序。

4.6.1.6 在非 Web 应用中，优雅地关闭 Spring IoC 容器

> 注意
>
> 本节仅对非 Web 应用来说。在相关 Web 应用程序关闭时，Spring 的基于 Web 的 ApplicationContext 实现已经有代码来优雅地关闭 Spring 的 Ioc 容器了。

如果你在非 Web 应用环境中使用 Spring 的 IoC 容器；比如，在富客户端桌面环境中； 你在 JVM 中注册了一个关闭的钩子。这么来做就保证了优雅地关闭并且在单例 bean 中调用相关销毁方法，那么所有资源都会被释放。当然，你必须正确配置并实现这些销毁回调方法。 要注册 一个 关闭 钩子， 可以 调用 AbstractApplicationContext 类中 声明的registerShutdownHook()方法：

```
import org.springframework.context.support.AbstractApplicationCo
ntext; import org.springframework.context.support.ClassPathXmlAp
plicationConte xt;
public final class Boot {
        public static void main(final String[] args) throws Exce
ption { AbstractApplicationContext ctx
            = new ClassPathXmlApplicationContext(new String []{"
beans.xml"});
        // 对上面的上下文添加一个关闭的钩子...
        ctx.registerShutdownHook();
        // 这里运行应用程序...
        // 主方法退出,钩子在应用关闭前优先被调用...
    }
}
```

4.6.2 ApplicationContextAware 和 BeanNameAware

当 ApplicationContext 创 建 实 现 了 org.springframework.context.ApplicationContextAware 接口的类时，这个类 就被提供了一个 ApplicationContext 的引用。

```
public interface ApplicationContextAware {
    void setApplicationContext(ApplicationContext applicationCon
text) throws BeansException;
}
```

因此 bean 就可以编程来操作创建 它们的 ApplicationContext ，通过 ApplicationContext 接口，或者转换到这个接口的已知子 类型，比如 ConfigurableApplicationContext，会公开更多的功能。一种使用方式就是编程式获取其它的 bean。有时候这种能力是很有用的。然而，通常你应该避免这么来做，因为它会耦合代码到 Spring 中，还没有协作者作为属性提供给 bean 的控制反转的风格。应用上下文 的其它方法提供了访问文件资源，公共应用事件和访问消息资源的方法。这些补充特性为在 4.14 节，"应用上下文的补充功能"来说明。

在 Spring 2.5 中，自动装配是获取 ApplicationContext 引用的另外一种方式。"传统的"constructor 和 byType 自动装配模式（在 4.4.5 节，"自动装配协作者"中说明的） 可以分别为构造方法参数或 setter 方法参数提供 ApplicationContext 类型的依赖。为了更大的灵活性，包括自动装配字段和多参数方法的能力，还有使用新的基于注解的自动装 配特性。如果字段，构造方法或者普通方法进行@Autowired 注解，而且你这么做了， ApplicationFactory 就会自动装配到期望 BeanFactory 类型的字段，构造方法参数或方法参数。要获取更多信息，可以参考 4.9.2 节，"@Autowired 和@Inject"。

当应用上下文创建实现了 org.springframework.beans.factory.BeanNameAware 接口的类时，这个类会被 提供一个在相关对象中定义的命名引用。

```
public interface BeanNameAware {
    void setBeanName(string name) throws BeansException;
}
```

回调函数在普通 bean 的属性被填充之后并且在如 InitializingBean 的 afterPropertiesSet 或自定义初始化方法的初始化回调之前被调用。

4.6.3 其它 Aware 接口

除了上面讨论的 ApplicationContextAware 和 BeanNameAware，Spring 还提供了很多 Aware 接口，它们允许 bean 来告诉容器它们需要确定的基础的依赖。最重要的 Aware 接口在下面总结出来了-作为一个通用的规则，名称就很好地说明了依赖的类型：

表 4.4 Aware 接口

名称	注入的依赖	何处解释
ApplicationContextA ware	声明 ApplicationContext	4.6.2 节，" ApplicationContextAware 和 BeanNameAware"

ApplicationEventPublisherAware	包含在 ApplicationContext 内的事件发布	4.14 节，“应用上下文的补充 功能”
BeanClassLoaderAware	用于加载 bean 类的类加载器	4.3.2 节，“实例化 bean”
BeanFactoryAware	声明 BeanFactory	4.6.2 节，“ ApplicationContextAware 和 BeanNameAware”
BeanNameAware	声明 bean 的名称	4.6.2 节，“ ApplicationContextAware 和 BeanNameAware”
BootstrapContextAwa re	容器运行的资源适配器 BootstrapContext。典型地是仅仅在 JCA 感知的 ApplicationContext 中可用。	第 24 章，JCA CCI
LoadTimeWeaverAware	在加载时为处理类定义的织入 器	8.8.4 节，“Spring Framework中使用 Aspect J 进行加载时织入”
MessageSourceAware	解析消息配置的策略（支持参数化和国际化）	4.14 节，“应用上下文的补充功能”
NotificationPublisherAware	Spring JMX 通知发布	23.7 节，“通知”
PortletConfigAware	容器运行的当前的PortletConfig。仅在感知 Web 的 Spring 的 ApplicationContext 中有 效	第 19 章，Portlet MVC 框架
PortletContextAware	容器运行的当前的PortletContext。仅在感知 Web 的 Spring 的 ApplicationContext 中有 效	第 19 章，Portlet MVC 框架

ResourceLoaderAware	为低级别的资源访问配置的加载器	第 5 章，资源
ServletConfigAware	容器运行的当前的ServletConfig。仅在感知 Web 的 Spring 的 ApplicationContext 中有效	第 16 章，Web MVC 框架
ServletContextAware	容器运行的当前的ServletContext。仅在感知 Web 的 Spring 的 ApplicationContext 中有 效	第 16 章，Web MVC 框架

请再次注意绑定你的代码到 Spring API 的这些接口的使用，并不再遵循控制反转风格。

因此，它们被推荐使用到基础的 bean 中，那些 bean 需要编程式地访问容器。

4.7 Bean 定义的继承

bean 的定义可以包含很多配置信息包括构造方法参数，属性值和容器指定的信息，比 如初始化方法，静态工厂方法名称等。子 bean 定义继承从父 bean 中获得的配置元数据。 子 bean 可以覆盖一些值或者添加其它所需要的。使用父子 bean 定义可以节省很多输入。 实际上，这是一种模板形式。

如 果 你 编 程 式 地 使 用 ApplicationContext 接口，子 bean 的 定 义 可 以 由 ChildBeanDefinition 类代 表。 很多 用户 不使 用这 个级 别的 方法 ，而 是在 类似于 ClassPathXmlApplicationContext.中声明式地配置 bean 的信息。当使用基于 XML 的配置元数据时，你可以使用 parent 属性来标识一个子 bean，使用这个属性的值来标识 父 bean。

```
<bean id="inheritedTestBean" abstract="true" class="org.springfr
amework.beans.TestBean">
<property name="name" value="parent"/>
<property name="age" value="1"/>
</bean>
<bean id="inheritsWithDifferentClass" class="org.springframework
.beans.DerivedTestBean" parent="inheritedTestBean" init-method="
initialize">
<property name="name" value="override"/>
<!-- age属性值1将会从父bean中继承过来 -->
</bean>
```

如果没有指定一个子 bean 使用父 bean 的类，但也可以覆盖它。在这种情形中，子 bean的类必须和父 bean 兼容，也就是说，它必须接受父 bean 的属性值。

子 bean 的定义继承构造方法参数值，属性值，还有父 bean 的方法覆盖，添加新值的 选择。任何你指定的初始化方法，销毁方法，和/或 static 工厂方法设置会覆盖对应父 bean 中的设置。

剩下的设置通常是从子 bean 来定义：依赖，自动装配模式，依赖检测，单例，范围， 延迟初始化。

前面的示例明确地使用了 abstract 属性来标识了父 bean 的定义。如果父 bean 没有 指定类，那么明确地标识父 bean 就必须要有 abstract，如下所示：

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>
<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean" parent="inheritedTestBeanWithoutClass"
    init-method="initialize">
    <property name="name" value="override"/>
    <!-- age属性值1将会从父bean中继承过来 -->
</bean>
```

父 bean 不会被实例化，因为它自己是不完整的，而且也明确地被 abstract 标记。当 bean 的定义是 abstract 这样的，那么它也仅仅被用做纯 bean 定义的模板，作为子 bean 定义的父 bean。尝试使用这种自身是 abstract 的父 bean，作为另外一个 bean 参考的属 性来指代它，或者使用父 bean 的 id 来明确使用 getBean()方法调用，会返回错误。相似 地，容器内部的 preInstantiateSingletons()方法忽略了抽象 bean 的定义。

> 注意
>
> 默认情况下，预实例化所有单例 bean。因此，如果你有仅仅想用作模板的（父）bean， 而且这个 bean 指定了一个类，那么必须将 abstract 属性设置为 true，这点是很重要的。否 则，应用上下文就会（尝试）预实例化 abstract 的 bean。

4.8 容器扩展点

通常情况下，应用程序开发人员不需要编写 ApplicationContext 实现类的子类。 相反，Spring 的 IoC 容器可以通过插件的方式来扩展，就是实现特定的整合接口。下面的几 个章节会来说明这些整合接口。

4.8.1 使用 BeanPostProcessor 来自定义 bean

BeanPostProcessor 接口定义了你可以提供实现你自己的（或覆盖容器默认的）实例逻辑，依赖解析逻辑等的回调方法。如果你想在 Spring 容器完成实例化，配置和初始化 bean 之后实现一些自定义逻辑，那么你可以使用一个或多个 BeanPostProcessor 实现类 的插件。

你可以配置多个 BeanPostProcessor 实例，而且你还可以通过设置 order 属性来控制这些 BeanPostProcessor 执行的顺序。 仅当 BeanPostProcessor 实现了 Ordered 接口你才可以设置设个属性；如果你想编写你自己的 BeanPostProcessor，你 也应该考虑实现 Ordered 接 口 。 要 了 解 更 多 细 节 ，可 以 参 考 JavaDoc 文档中的 BeanPostProcessor 和 Ordered 接口。

> 注意
>
> BeanPostProcessor 操作 bean（对象）实例；那也就是说，Spring 的 IoC 容器 实例化 bean 的实例，之后 BeanPostProcessor 来做它们要做的事情。

BeanPostProcessor 的范围是对于每一个容器来说的。如果你使用了容器继承，那么这才有所关联。如果你在一个容器中定义了 BeanPostProcessor，那么它仅仅 会在那个容器中后处理 bean。换句话说，一个容器中定义的 bean 不会被另外一个容器 中定义的 BeanPostProcessor 来进行后处理，即便是两个容器都是相同继承链上的 一部分。

要修改真正的 bean 定义（也就是说，定义 bean 的蓝图），你可以使用 4.8.2 节，"使 用 BeanFactoryPostProcessor 来 自 定 义 配 置 元 数 据" 描 述 的 BeanFactoryPostProcessor 来进行。
org.springframework.beans.factory.config.BeanPostProcessor 接口由两个回调方法构成。如果在容器中有一个类注册为后处理器，对于容器创建的每个 bean 的实例，后处理器从容器中获得回调方法，在容器初始化方法之前（比如 InitializingBean 的 afterPropertiesSet()方法和任意声明为初始化的方法）被调用，还有在 bean 初始化回调之后 被调用。后处理器可以对 bean 实例采取任何动作，包括完整忽略回调。通常来说 bean 的后处理器会对回调接口进行检查，或者会使用代理包装 bean。一些 Spring 的 AOP 基类也会 作为 bean 的后处理器实现来提供代理包装逻辑。

ApplicationContext 会自动检测任意实现了 BeanPostProcessor 接口的 bean 定义的配置元数据。ApplicationContext 注册这些 bean 作为后处理器，那么它们可以在 bean 创建之后被调用。Bean 的后处理器可以在容器中部署，就像其它 bean 那

样。

BeanPostProcessor 和 AOP 自动代理

实现了 BeanPostProcessor 接口的类是特殊的，会被容器不同对待。所有它们参照的 BeanPostProcessor 和 bea n 会在启动时被实例化，作为

ApplicationContext 启动阶段特殊的一部分。接下来，所有的 BeanPostProcessor 以排序的方式注册并应用于容器中的其它 bean。因为 AOP 自动 代理作为 BeanPostProcessor 本身的实现，它们为自动代理资格的直接引用的既不是 BeanPostProcessor 也不是 bean，因此没有织入它们的方面。

对于这样的 bean，你应该看到一个信息级的日志消息：“Bean foo 没有由所有 BeanPostProcessor 接口处理的资格（比如：没有自动代理的资格）”。 下面的示例展示了如何在 ApplicationContext 中编写，注册和使用BeanPostProcessor。

4.8.1.1 示例：BeanPostProcessor 风格的 Hello World

第一个示例说明了基本的用法。示例展示了一个自定义的 BeanPostProcessor 实现类来调用每个由容器创建 bean 的 toString()方法并打印出字符串到系统的控制台中。

自定义 BeanPostProcessor 实现类的定义：

```
package scripting;
import org.springframework.beans.factory.config.BeanPostProcesso
r;
import org.springframework.beans.BeansException;
public class InstantiationTracingBeanPostProcessor implements
BeanPostProcessor {
    // 简单地返回实例化的bean
    public Object postProcessBeforeInitialization(Object bean, S
tring beanName)throws BeansException {
        return bean; // 这里我们可能返回任意对象的引用...
    }
    public Object postProcessAfterInitialization(Object bean, St
ring beanName)throws BeansException {
        System.out.println("Bean '" + beanName + "' created : "
+
        bean.toString());
        return bean;
    }
}
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:lang="ht
tp://www.springframework.org/schema/lang" xsi:schemaLocation="ht
tp://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring -beans-3.0.xsd http://www.springfra
mework.org/schema/lang http://www.springframework.org/schema/lan
g/spring-lang-3.0.xsd">
    <lang:groovy id="messenger"
        script-source="classpath:org/springframework/scripting/g
r oovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just
 So
        Dreamy."/>
    </lang:groovy>
    <!--
    当上面的bean(messenger)被实例化时，这个自定义的BeanPostProcessor实
 现会输出实际内容到系统控制台
    -->
    <bean
    class="scripting.InstantiationTracingBeanPostProcessor" />
</beans>
```

注意 InstantiationTracingBeanPostProcessor 仅仅是简单地定义。它也没有 命名，因为它会像其它 bean 那样被依赖注入。（前面的配置也可以定义成 Groovy 脚本支持 的 bean。Spring 2.0 动态语言支持在第 27 章中来详细说明）

下面示例的 Java 应用程序执行了前面配置的代码：

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicati
onConte xt;
import org.springframework.scripting.Messenger;
public final class Boot {
    public static void main(final String[] args) throws Exceptio
n {
        ApplicationContext ctx = new
            ClassPathXmlApplicationContext("scripting/beans.xml"
);
        Messenger messenger = (Messenger) ctx.getBean("messenger
"); System.out.println(messenger);
    }
}
```

上面应用程序的输出类似于如下内容：

```
Bean 'messenger' created : org.springframework.scripting.groovy.
GroovyMessenger@272961 org.springframework.scripting.groovy.Groo
vyMessenger@272961
```

4.8.1.2 示例：RequiredAnnotationBeanPostProcessor

配合自定义的 BeanPostProcessor 实现使用回调接口或者注解，是扩展 Spring IoC 容器的一种通用方式。Spring 的 RequiredAnnotationBeanPostProcessor 就是一个例子 – 一种 BeanPostProcessor 实现，随着 Spring 一起发布，来保证 bean 中被（任意）注解所标记的 JavaBean 属性在真正（配置）注入时有值。

4.8.2 使用 BeanFactoryPostProcessor 自定义配置元数据

下一个扩展点我们要来看看 org.springframework.beans.factory.config.BeanFactoryPostProcessor 。这个接口的语义和那些 BeanPostProcessor 是相似的，但有一个主要的不同点：BeanFactoryPostProcessor 操作 bean 的配置元数据；也就是说 Spring 的 IoC 容器允许 BeanFactoryPostProcessor 来读取配置元数据并在容器实例化 BeanFactoryPostProcessor 以外的任何 bean 之前可以修改它。

你可以配置多个 BeanFactoryPostProcessor，并且你也可以通过 order 属性来控制这些 BeanFactoryPostProcessor 执行的顺序。然而，仅当 BeanFactoryPostProcessor 实现 Ordered 接口时你才能设置这个属性。如果编写你自己的 BeanFactoryPostProcessor，你也应该考虑实现 Ordered 接口。参考 JavaDoc 文档来获取 BeanFactoryPostProcessor 和 Ordered 接口的更多细节。

> 注意
>
> 如果你想改变真实的 bean 实例（也就是说，从配置元数据中创建的对象），那么你 需要使用 BeanPostProcessor（在上面 4.8.1 节，“使用 BeanPostProcessor 来自定义 bean ”中描述）来代替。在 BeanFactoryPostProcessor （比如使用 BeanFactory.getBean()）中来使用这些 bea n 的实例虽然在技术上是可行的，但 这么来做会引起 bean 过早实例化，违反标准的容器生命周期。这也会引发一些副作用， 比如绕过 bean 的后处理。

而且，BeanFactoryPostProcessor 的范围也是对每一个容器来说的。如果你 使 用 了容 器的 继承 的话 ，这 就是 唯一 相关 的点 了。 如果 你在 一个 容器 中定 义 了 BeanFactoryPostProcessor，那么它只会用于在那个容器中的 bean。一个容器中 Bean 的定义不会被另外一个容器中的 BeanFactoryPostProcessor 后处理，即使 两个容器都是相同继承关系的一部分。

当在 ApplicationContext 中声明时，bean 工厂后处理器会自动被执行，这就可以对定义在容器中的配置元数据进行修改。Spring 包含了一些预定义的 bean 工厂后处理器，比如 PropertyOverrideConfigurer 和 PropertyPlaceholderConfigurer.。自定义的 BeanFactoryPostProcessor 也可以来用，比如，注册自定义的属性编辑器。

ApplicationContext 会 自 动 检 测 任 意 部 署 其 中 ， 且 实 现 了 BeanFactoryPostProcessor 接口的 bean。在适当的时间，它用这些 bean 作为 bean 工厂后处理器。你可以部署这些后处理器 bean 作为你想用的任意其它的 bean。

> 注意
>
> 和 BeanPostProcessor 一样，通常你不会想配置 BeanFactoryPostProcessor 来进行延迟初始化。如果没有其它 bean 引用 Bean(Factory)PostProcessor，那么后 处理器就不会被初始化了。因此，标记它为延迟初始化就会被忽略，即便你在 `<beans/>` 元 素声明中设置 default-lazy-init 属性为 true，那么 Bean(Factory)PostProcessor 也会正常被初始化。

4.8.2.1 示例：PropertyPlaceholderConfigurer

你可以使用来对使用了标准 Java Properties 格式的分离文件中定义的 bean 来声明属 性值。这么来做可以使得部署应用程序来自定义指定的环境属性，比如数据库的连接 URL 和密码，不会有修改容器的主 XML 定义文件或其它文件的复杂性和风险。

考虑一下下面这个基于 XML 的配置元数据代码片段，这里的 dataSource 就使用了占位符来定义。这个示例展示了从 Properties 文 件 中 配 置 属 性 的 方 法 。 在 运 行 时 ， PropertyPlaceholderConfigurer 就会用于元数据并为数据源替换一些属

性。指定替 换的值作为${属性-名}形式中的占位符，这里应用了 Ant/log4j/JSP EL 的风格。

```
<bean class="org.springframework.beans.factory.config.PropertyPl
aceho lderConfigurer">
    <property name="locations" value="classpath:com/foo/jdbc.pro
perties"/>
</bean>
<bean id="dataSource" destroy-method="close" class="org.apache.c
ommons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassNa
me}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

而真正的值是来自于标准的 Java Properties 格式的文件：

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

因此，字符串${jdbc.username}在运行时会被值'sa'替换，对于其它占位符来说也是相同的，匹配到了属性文件中的键就会用其值替换占位符。PropertyPlaceholderConfigurer 在很多 bean 定义的属性中检查占位符。此外，对占位符可以自定义前缀和后缀。

使用 Spring 2.5 引入的 context 命名空间，也可以使用专用的配置元素来配置属性占 位符。在 location 属性中，可以提供一个或多个以逗号分隔的列表。

```
<context:property-placeholder
    location="classpath:com/foo/jdbc.properties"/>
```

PropertyPlaceholderConfigurer 不仅仅查看在 Properties 文件中指定的属 性。默认情况下，如果它不能在指定的属性文件中发现属性，它也会检查 Java System 属性。 你可以通过设置 systemPropertiesMode 属性，使用下面整数的三者之一来自定义这种 行为：

never(0)：从不检查系统属性

fallback(1)：如果没有在指定的属性文件中解析到属性，那么就检查系统属性。这是默 认的情况。

override(2)：在检查指定的属性文件之前，首先去检查系统属性。这就允许系统属性覆 盖其它任意的属性资源。

查看 PropertyPlaceholderConfigurer 的 JavaDoc 文档来获取更多信息。

类名替换

你可以使用 PropertyPlaceholderConfigurer 来替换类名，在运行时，当你不得不去选择一个特定的实现类时，这是很有用的。比如：

```
<bean class="org.springframework.beans.factory.config.PropertyPl
ac eholderConfigurer">
    <property name="locations">
        <value>classpath:com/foo/strategy.properties< /value>
    </property>
    <property name="properties">
        <value>custom.strategy.class=com.foo.DefaultStrategy< /
value>
    </property>
</bean>
<bean id="serviceStrategy" class="${custom.strategy.class}"/>
```

如果类在运行时不能解析成一个有效的类，那么在即将创建时，bean 的解析就失败了，这是 ApplicationContext 在对非延迟初始化 bean 的 preInstantiateSingletons()阶段发生的。

4.8.2.2 示例：PropertyOverrideConfigurer

PropertyOverrideConfigurer，另外一种 bean 工厂后处理器，类似于 PropertyPlaceholderConfigurer，但不像后者，对于所有 bean 的属性，原始定义可 以有默认值或没有值。如果一个 Properties 覆盖文件没有特定 bean 的属性配置项，那么 就会使用默认的上下文定义。

注意，bean 定义是不知道被覆盖的，所以从 XML 定义文件中不能立即明显反应覆盖配 置。在多个 PropertyOverrideConfigurer 实例的情况下，为相同 bean 的属性定义不同的值，那么最后一个有效，这就是覆盖机制。 属性文件配置行像这种格式：

```
beanName.property=value
```

比如：

```
dataSource.driverClassName=com.mysql.jdbc.Driver dataSource.url=
jdbc:mysql:mydb
```

这个示例文件可以用于包含了 dataSource bean 的容器，它有 driver 和 url 属性。复合属性名也是支持的，除了最终的属性被覆盖，只要路径中的每个组件都是非空的（假设由构造方法初始化）。在这个例子中...

```
foo.fred.bob.sammy=123
```

...foo bean 的 fred 属性的 bob 属性的 sammy 属性的值设置为标量 123。 注意指定的覆盖值通常是文字值；它们不会被翻译成 bean 的引用。当 XML 中的 bean 定义的原始值指定了 bean 引用时，这个约定也适用。

使用 Spring 2.5 引入的 context 命名空间，可以使用专用的配置元素来配置属性覆盖：

```
<context:property-override
location="classpath:override.properties"/>
```

4.8.3 使用 FactoryBean 来自定义实例化逻辑

实现了 org.springframework.beans.factory.FactoryBean 接口的对象它们 就是自己的工厂。

FactoryBean 接口就是 Spring IoC 容器实例化逻辑的可插拔点。如果你的初始化代码 很复杂，那么相对于（潜在地）大量详细的 XML 而言，最好是使用 Java 语言来表达。你可 以 创 建 自己 的 FactoryBean ，在类中编写复杂的初始化代码，之后将你自定义的 FactoryBean 插入到容器中。

FactoryBean 接口提供下面三个方法：

Object getObject()：返回工厂创建对象的实例。这个实例可能被共享，那就是看这个工厂返回的是单例还是原型实例了。

boolean isSingleton()：如果 FactoryBean 返回单例的实例，那么该方法返回 true， 否则就返回 false。

Class getObjectType()：返回由 getObject()方法返回的对象类型，或者事先不知 道类型时返回 null。

FactoryBean 的概念和接口被用于 Spring Framework 中的很多地方；随 Spring 发行， 有超过 50 个 FactoryBean 接口的实现类。

当你需要向容器请求一个真实的 FactoryBean 实例，而不是它生产的 bean，当调用 ApplicationContext 的 getBean()方法时，在 bean 的 id 之前要有连字符（&）。所以 对于一个给定 id 为 myBean 的 FactoryBean，调用容器的 getBean("myBean")方法返回的 FactoryBean 产品；而调用 getBean("&myBean")方法则返回 FactoryBean 实例本身。

4.9 基于注解的容器配置

配置Spring时，注解要比XML更好吗？ 基于注解配置的介绍就提出了这样的问题，这种方法要比XML'更好'吗？简短的回答就是具体问题具体分析。完整的答案就是每种方法都有它的利与弊，通常是让开发 人员来决定使用哪种策略更适合使用。由于定义它们的方式，注解在它们的声明中提供了大量的上下文，使得配置更加简短和简洁。然而，XML更擅长装配组件，而不需要触碰它们源代码或重新编译。一些开发人员更喜欢装配源码而其他人认为被注解的类不再 是POJO了，此外，配置变得分散并且难以控制。

无论怎么去线则，Spring都可以容纳两种方式，甚至是它们的混合体。最值得指出的是通过JavaConfig（4.12节）选择，Spring允许以非侵入式的方式来使用注解，而不需 要触碰目标组件的源代码和工具，所有的配置方式都是SpringSource Tool Suite所支持的。

作为 XML 配置的另外一种选择，依靠字节码元数据的基于注解的配置来装配组件代替 了尖括号式的声明。作为使用 XML 来表述 bean 装配的替换，开发人员可以将配置信息移入 到组件类本身中，在相关的类，方法或字段声明上使用注解。正如在 4.8.1.2 节，"示例： RequiredAnnotationBeanPostProcessor"所提到的，使用 BeanPostProcessor 来连接注解是扩展 Spring IoC 容器的一种常用方式。比如，Spring 2.0 引入的使用@Required（4.9.1 节）注解来强制所需属性的可能性。在 Spring 2.5 中，可以使用相同的处理方法来驱 动 Spring 的依赖注入。从本质上来说，@Autowired 注解提供了在 4.4.5 节，"自动装配协 作者"中描述的相同能力，但却有更细粒度的控制和更广泛的适用性。Spring 2.5 也添加了 对 JSR-250 注解的支持，比如@Resource，@PostConstruct 和@PreDestroy。Spring 3.0 添加了对 JSR-330（对 Java 的依赖注入）注解的支持，包含在 javax.inject 包下，比如@Inject，@Qualifier，@Named 和@Provider，当 JSR330 的 jar 包在类路径下时就可以使用。使 用这些注解也需要在 Spring 容器中注册特定的 BeanPostProcessor。

> 注意
>
> 注解注入会在 XML 注入之前执行，因此通过两种方式，那么后面的配置会覆盖前面装 配的属性。

一如往常，你可以注册它们作为独立的 bean，但是它们也可以通过包含下面的基于 XML的 Spring 配置代码片段被隐式地注册（注意要包含 context 命名空间）：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:context=
"http://www.springframework.org/schema/context" xsi:schemaLocati
on="http://www.springframework.org/schema/bean[s http://www.spri
ngframework.org/schema/beans/spring -beans-3.0.xsd http://www.sp
ringframework.org/schema/context http://www.springframework.org/
schema/context/spring-context-3.0.xsd">
    <context:annotation-config/>
</beans>
```

（隐式注册的后处理器包含 AutowiredAnnotationBeanPostProcessor，CommonAnnotationBeanPostProcessor，PersistenceAnnotationBeanPostProcessor，以及上述的 RequiredAnnotationBeanPostProcessor）

> 注意
>
> `<context:annotation-config/>` 仅仅查找定义在同一上下文中的 bean 的注解。 这就意味着，如果你为 DispatcherServlet 将 `<context:annotation-config/>` 放置在 WebApplicationContext 中，那么它仅仅检查控制器中的@Autowired bean，而 不是你的服务层 bean，可以参看 16.2 节，“DispatcherServlet”来查看更多信息。

4.9.1 @Required

@Required 注解应用于 bean 属性的 setter 方法，就向下面这个示例：

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

这个注解只是表明受影响的 bean 的属性必须在 bean 的定义中或者是自动装配中通过明确的属性值在配置时来填充。如果受影响的 bean 属性没有被填充，那么容器就会抛出异 常；这就允许了急切而且明确的失败，要避免 NullPointerException。我们推荐你放 置断言到 bean 的类中，比如，放置到初始化方法中。当你在容器外部使用类时，这么来做 是强制那些所需的引用和值。

4.9.2 @Autowired 和@Inject

正如预期的那样，你可以使用@Autowired 注解到“传统的”setter 方法中： 注意在下面的示例中，JSR330 的@Inject 注解可以用于代替 Spring 的@Autowired。没有必须的属性，不像 Spring 的@Autowired 注解那样，如果要注入的值是可选的话，要有一个 required 属性来表示。如果你将 JSR330 的 JAR 包放置到类路径下的话，这种行为就会自 动开启。

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

你也可以将注解应用于任意名称和/或多个参数的方法：

```
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public void prepare(MovieCatalog movieCatalog, CustomerPrefe
renceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

你也可以将用于构造方法和字段：

```
public class MovieRecommender {
    @Autowired
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPrefer
enceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

也可以提供 ApplicationContext 中特定类型的所有 bean，通过添加注解到期望哪种类型的数组的字段或者方法上：

```
public class MovieRecommender {
    @Autowired
    private MovieCatalog[] movieCatalogs;
    // ...
}
```

相同地，也可以用于特定类型的集合：

```
public class MovieRecommender {
    private Set<MovieCatalog> movieCatalogs;
    @Autowired
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs
) {
        this.movieCatalogs = movieCatalogs;
    }
    // ...
}
```

甚至特定类型的 Map 也可以自动装配，但是期望的键的类型是 String 的。Map 值会 包含所有期望类型的 bean，而键会包含对应 bean 的名字：

```
public class MovieRecommender {
    private Map<String, MovieCatalog> movieCatalogs;
    @Autowired
    public void setMovieCatalogs(Map<String, MovieCatalog> movie
Catalogs) {
        this.movieCatalogs = movieCatalogs;
    }
    // ...
    }
```

默认情况下，当出现零个候选 bean 的时候，自动装配就会失败；默认的行为是将被注解的方法，构造方法和字段作为需要的依赖关系。这种行为也可以通过下面这样的做法来改变。

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired(required=false)
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

> 注意
>
> 每一个类中仅有一个被注解方法可以标记为必须的，但是多个非必须的构造方法可 以被注解。在那种情况下，每个构造方法都要考虑到而且 Spring 使用依赖可以被满足 的那个构造方法，那就是参数最多的那个构造方法。

@Autowired 需要的属性推荐使用@Required 注解。所需的表示了属性对于自动 装配目的不是必须的，如果它不能被自动装配，那么属性就会忽略了。另一方面，

@Required 更健壮一些，它强制了由容器支持的各种方式的属性设置。如果没有注入任何值，就会抛出对应的异常。 你也可以针对我们熟知的解决依赖关系的接口来使用@Autowired：BeanFactory，ApplicationContext ， Environment ，ResourceLoader ，ApplicationEventPublisher 和 MessageSource。这些接口和它们的扩展接口，比 如 ConfigurableApplicationContext 或 ResourcePatternResolver 也会被自动解析，而不需要特殊设置的必要。

```
public class MovieRecommender {
    @Autowired
    private ApplicationContext context;
    public MovieRecommender() {

    }
    // ...
}
```

> 注意
>
> @Autowired ， @Inject ， @Resource 和 @Value 注 解 是 由 Spring 的 BeanPostProcessor 实现类来控制的，反过来告诉你你不能在 BeanPostProcessor 或 BeanFactoryPostProcessor 类型（任意之一）应用这些注解。这些类型必须明确地 通过 XML 或使用 Spring 的@Bean 方法来‘装配’。

4.9.3 使用限定符来微调基于注解的自动装配

因为通过类型的自动装配可能导致多个候选者，那么在选择过程中通常是需要更多的控 制的。达成这个目的的一种做法就是 Spring 的@Qualifier 注解。你可以用特定的参数来 关联限定符的值，缩小类型的集合匹配，那么特定的 bean 就为每一个参数来选择。最简单 的情形，这可以是普通描述性的值：

> ⓘ
>
> 注意
>
> JSR 330 的@Qualifier 注解仅仅能作为元注解来用，而不像 Spring 的 @Qualifier 可以 使用字符串属性来在多个注入的候选者之间区别，并可以放在注解，类型，字段，方法，构 造方法和参数中。

```
public class MovieRecommender {
    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;
    // ...
}
```

@Qualifier 注解也可以在独立构造方法参数或方法参数中来指定：

```
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public void prepare(@Qualifier("main") MovieCatalog movieCat
alog,CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```

对应的 bean 的定义可以按如下所示。限定符值是“main”的 bean 会用限定了相同值的构造方法参数来装配。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:context=
"http://www.springframework.org/schema/context" xsi:schemaLocati
on="http://www.springframework.org/schema/beans http://www.sprin
gframework.org/schema/beans/spring -beans-3.0.xsd http://www.spr
ingframework.org/schema/contex)t http://www.springframework.org/
schema/context/spring-context-3.0.xsd">
    <context:annotation-config/>
    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/>
        <!-- 注入这个bean需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/>
        <!-- 注入这个bean需要的任何依赖 -->
    </bean>
    <bean id="movieRecommender" class="example.MovieRecommender"
/>
</beans>
```

对于后备匹配，bean 名称被认为是默认的限定符值。因此你可以使用 id 为“main”来 定义 bean，来替代嵌套的限定符元素，这也会达到相同的匹配结果。然而，尽管你使用这 种规约来通过名称参照特定的 bean，@Autowired 从根本上来说就是关于类型驱动注入和可 选语义限定符的。这就是说限定符的值，即便有 bean 的名称作为后备，通常在类型匹配时 也会缩小语义；它们不会在语义上表达对唯一 bean 的 id 的引用。好的限定符的值是“main” 或“EMEA”或“persistent”，特定组件的表达特性是和 bean 的 id 独立的，在比如前面示例 之一的匿名 bean 的情况下它是可能自动被创建的。

限定符也可以用于类型集合，正如上面讨论过的，比如 `Set<MovieCatalog>` 。在这 种情况下，根据声明的限定符，所有匹配的 bean 都会被注入到集合中。这就说明了限定符 不必是唯一的；它们只是构成了筛选条件。比如，你可以使用相同的限定符“action”来定 义多个 MovieCatalog bean；它们全部都会通过 @Qualifier("action")注解注入到 `Set<MovieCatalog>` 中。

> 提示
>
> 如果你想通过名称来表达注解驱动注入，主要是不使用@Autowired，即便在技术上来说能够通过@Qualifier 值指向一个 bean 的名称。相反 ，使用 JSR-250 的 @Resource 注解，在语义上定义了通过它的唯一名称去确定一个具体的目标组件，声 明的类型和匹配过程无关。

由于这种语义区别的特定后果，bean 本身被定义为集合或 map 类型，是不能通过 @Autowired 来进行注入的，因为类型匹配用于它们不太合适。对于这样的 bean 使用 @Resource，通过唯一的名称指向特定的集合或 map 类型的 bean。

@Autowired 可以用于字段，构造方法和多参数方法，在参数级允许通过限定符 注解缩小。相比之下，@Resource 仅支持字段和仅有一个参数的 bea n 属性的 setter 方法。因此，如果你的注入目标是构造方法或多参数的方法，那么就坚持使用限定符。 你可以创建你自定义的限定符注解。只需定义一个注解并在你的定义中提供 @Qualifier 注解即可。 注意你可以以下面描述的方式来使用 JSR 330 的 @Qualifier 注解，用于替换 Spring 的 @Qualifier 注解。如果在类路径中有 JSR 330 的 jar 包，那么这种行为是自动开启的。

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre { String value(); }
```

之后你可以在自动装配的字段和参数上提供自定义的限定符：

```
public class MovieRecommender {
    @Autowired
    @Genre("Action")
    private MovieCatalog actionCatalog;
    private MovieCatalog comedyCatalog;
    @Autowired
    public void setComedyCatalog(@Genre("Comedy") MovieCatalog c
omedyCatalog) {
        this.comedyCatalog = comedyCatalog;
    }
    // ...
}
```

之后，为候选 bean 提供信息，你可以添加 `<qualifier/>` 标签来作为 `<bean/>` 标签的 子元素，然后指定 type 和 value 值来匹配你自定义的限定符注解。这种类型匹配是基于 注解类的完全限定名。否则，作为一种简便的形式，如果没有名称冲突存在的风险，你可以 使用短类名。这两种方法都会在下面的示例中来展示。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:context=
"http://www.springframework.org/schema/context" xsi:schemaLocati
on="http://www.springframework.org/schema/beans http://www.sprin
gframework.org/schema/beans/spring -beans-3.0.xsd
http://www.springframework.org/schema/context http://www.springf
ramework.org/schema/context/spring-context-3.0.xsd">
    <context:annotation-config/>
    <bean class="example.SimpleMovieCatalog">
        <qualifier type="Genre" value="Action"/>
        <!— 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <qualifier type="example.Genre" value="Comedy"/>
        <!— 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean id="movieRecommender" class="example.MovieRecommender"
/>
</beans>
```

在 4.10 节，“类路径扫描和管理的组件”中，你会看到基于注解的替代，在 XML 中来 提供限定符元数据。特别是在 4.10.7 节，“使用注解提供限定符元数据”中。

在一些示例中，使用无值的注解就足够了。这当注解服务于多个通用目的时是很有用的，

而且也可以用于集中不同类型的依赖关系。比如，当没有因特网连接时，你可以提供脱机目 录来由于搜索。首先定义简单的注解：

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Offline {
}
```

之后将注解添加到要被自动装配的字段或属性上：

```
public class MovieRecommender {
    @Autowired
    @Offline
    private MovieCatalog offlineCatalog;
    // ...
}
```

现在来定义 bean 的限定符的 type：

```
<bean class="example.SimpleMovieCatalog">
<qualifier type="Offline"/>
<!— 注入这个bean所需要的任何依赖 -->
</bean>
```

你也可以定义自定义的限定符注解来接受命名属性，除了或替代简单的 value 属性。 如果多个属性值之后在要不自动装配的字段或参数上来指定，那么要考虑 bean 的定义必须 匹配自动装备候选者的所有属性值。示例，考虑下面的注解定义：

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MovieQualifier { String genre();
Format format();
}
```

在本例中，Format 是枚举类型：

```
public enum Format { VHS, DVD, BLURAY }
```

要自动装配的字段使用自定义的限定符和包含 genre 和 format 两个属性的值来注解。

```
public class MovieRecommender {
    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Action")
    private MovieCatalog actionVhsCatalog;
    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Comedy")
    private MovieCatalog comedyVhsCatalog;
    @Autowired
    @MovieQualifier(format=Format.DVD, genre="Action")
    private MovieCatalog actionDvdCatalog;
    @Autowired
    @MovieQualifier(format=Format.BLURAY, genre="Comedy")
    private MovieCatalog comedyBluRayCatalog;
    // ...
}
```

最后，bean 的定义应该包含匹配的限定符值。这个示例也展示了 bean 的元属性可能用 于替代 `<qualifier/>` 子元素。如果可用， `<qualifier/>` 和它的属性优先，但是如果目 前没有限定符，自动装配机制就会在 `<meta/>` 标签提供的值上失效，就像下面这个示例中的 最后两个 bean。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:context=
"http://www.springframework.org/schema/context" xsi:schemaLocati
on="http://www.springframework.org/schema/beans http://www.sprin
gframework.org/schema/beans/spring -beans-3.0.xsd http://www.spr
ingframework.org/schema/context http://www.springframework.org/s
chema/context/spring-context-3.0.xsd">
    <context:annotation-config/>
    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Action"/>
        </qualifier>
        <!— 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Comedy"/>
        </qualifier>
        <!— 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="DVD"/>
        <meta key="genre" value="Action"/>
        <!— 注入这个bean所需要的任何依赖 -->
    </bean>
    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="BLURAY"/>
        <meta key="genre" value="Comedy"/>
        <!— 注入这个bean所需要的任何依赖 -->
    </bean>
</beans>
```

4.9.4 CustomAutowireConfigurer

CustomAutowireConfigurer 是 BeanFactoryPostProcessor 的一种，它使得 你可以注册你自己定义的限定符注解类型，即便它们没有使用 Spring 的@Qualifier 注解也是可以的。

```
<bean id="customAutowireConfigurer" class="org.springframework.b
eans.factory.annotation.CustomAu
    towireConfigurer">
    <property name="customQualifierTypes">
        <set>
            <value>example.CustomQualifier</value>
        </set>
    </property>
</bean>
```

AutowireCandidateResolver 的特别实现类对基于 Java 版本的应用程序上下文激活。在 Java 5 之前的版本中，限定符注解是不支持的，因此自动装配候选者仅由每个 bean 定义中的 autowire-candidate 值来决 定，还有 在 `<beans/>` 元 素中的任 何可用的 default-autowire-candidates 模 式 也 是 可 以 的 。 在 Java 5 之 后 的版 本 中 ， @Qualifier 注解的存在还有任意使用 CustomAutowireConfigurer 注册的自定义注解也将发挥作用。

不管 Java 的版本，当多个 bean 作为自动装配的候选者，决定“主要”候选者的方式也 是相同的：如果候选者中一个 bean 的定义有 primary 属性精确地设置为 true，那么它就会被选择。

4.9.5 @Resource

Spring 也支持使用 JSR 250 的@Resource 注解在字段或 bean 属性的 setter 方法上的注 入。这在 Java EE 5 和 6 中是一个通用的模式，比如在 JSF 1.2 中管理的 bean 或 JAX-WS 2.0 端点。Spring 也为其所管理的对象支持这种模式。

@Resource 使用名称属性，默认情况下 Spring 解释这个值作为要注入的 bean 的名称。

换句话说，如果遵循 by-name 语义，正如在这个示例所展示的：

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Resource(name="myMovieFinder")
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

如果没有明确地指定名称，那么默认的名称就从字段名称或 setter 方法中派生出来。以字段为例，它会选用字段名称；以 setter 方法为例，它会选用 bean 的属性名称。所以下面 的示例中有名为“movieFinder”的 bean 通过 setter 方法来注入：

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Resource
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

> 注意
>
> 使用注解提供的名称被感知 CommonAnnotationBeanPostProcessor 的 ApplicationContext 解析成 bean 的名称。名称可以通过 JNDI 方式来解析，只要你明确 地配置了 Spring 的 SimpleJndiBeanFactory。然而，还是推荐你基于默认行为并仅使用 Spring 的 JNDI 查找能力来保存间接的水平。

在使用@Resource 并没有 明确指 定名称的 独占情况 下，和 @Autowired 相似，@Resource 发现主要类型匹配，而不是特定名称 bean 并解析了熟知的可解析的依赖关系： BeanFactory ， ApplicationContext ， ResourceLoader ，ApplicationEventPublisher，MessageSource 和接口。

因此，在下面的示例中，customerPreferenceDao 字段首先寻找名为 customerPreferenceDao 的 bean，之后回到匹配 CustomerPreferenceDao 的主类型。 “context”字段基于已知的可解析的依赖类型 ApplicationContext 注入。

```
public class MovieRecommender {
    @Resource
    private CustomerPreferenceDao customerPreferenceDao;
    @Resource
    private ApplicationContext context;
    public MovieRecommender() {

    }
    // ...
}
```

4.9.6 @PostConstruct 和@PreDestroy

CommonAnnotationBeanPostProcessor 不但能识别@Resource 注解，而且还能识别 JSR-250 生命周期注解。在 Spring 2.5 中引入，对这些注解的支持提供了在初始化回调（4.6.1.1 节）和销毁回调（4.6.1.2 节）中的另一种选择。只要 CommonAnnotationBeanPostProcessor 在 Spring 的 ApplicationContext 中注册，一个携带这些注解之一的方法就同时被调用了，和 Spring 生命周期接口方法或明确地声明回调方法相对应。在下面的示例中，在初始化后缓存会预先填充，在销毁后会清理。

```
public class CachingMovieLister {
    @PostConstruct
    public void populateMovieCache() {
        // 在初始化时填充movie cache...
    }
    @PreDestroy
    public void clearMovieCache() {
        // 在销毁后清理movie cache...
    }
}
```

> 注意
>
> 关于组合多个生命周期机制影响的细节内容，请参考 4.6.1.4 节，“组合生命周期机制”。

4.10 类路径扫描和管理的组件

本章中的大多数示例都使用了 XML 来指定配置元数据在 Spring 的容器中生产每一个 BeanDefinition。之前的章节（4.9 节，"基于注解的容器配置"）表述了如何通过代码级 的注解来提供大量的配置信息。尽管在那些示例中，"基础的"bean 的定义都是在 XML 文 件中来明确定义的，而注解仅仅进行依赖注入。本节来说明另外一种通过扫描类路径的方式 来隐式检测候选组件。候选组件是匹配过滤条件的类库，并有在容器中注册的对应的 bean 的定义。这就可 以不用 XML 来执行 bea n 的注册了 ，那么你就 可以使用注 解（比如 @Component），AspectJ 风格的表达式，或者是你子定义的过滤条件来选择那些类会有在容 器中注册的 bean。

> 注意
>
> 从 Spring 3.0 开始，很多由 Spring JavaConfig 项目提供的特性作为 Spring Framework 核心 的一部分了。这就允许你使用 Java 而不是传统的 XML 文件来定义 bean 了。看一看 @Configuration，@Bean，@Import 和 @DependsOn 注解的例子来使用它们的新特性。

4.10.1 @Component 和更多典型注解

在 Spring 2.0 版之后，@Repository 注解是任意满足它的角色或典型（比如熟知的数 据访问对象，DAO）库的类的标记。在这个标记的使用中，就是在 14.2.2 节，"表达式翻译" 中描述的表达式自动翻译。

Spring 2. 5 引 入 了 更 多 的 注 解 ： @Component ， @Service 和 @Controller 。

@Component 是对 Spring 任意管理 组件的通用刻 板。@Repository ，@Service 和 @Controller 是对更多的特定用例@Component 的专业化，比如，在持久层，服务层和 表现层。因此，你可以使用@Component 注解你的组件类，但是使用 @Repository， @Service 或@Controller 注解来替代的话，那么你的类更合适由工具来处理或和不同的方面相关联。比如，这些刻板注解使得理想化的目标称为切入点。而且@Repository， @Service 和@Controller 也可以在将来 Spring Framework 的发布中携带更多的语义。因此，如果对于服务层，你在 @Component 或@Service 中间选择的话，那么@Service 无 疑是更好的选择。相似地，正如上面提到的，在持久层中，@Repository 已经作为自动异 常翻译的表示所被支持

4.10.2 自动检测类和 bean 的注册

Spring 可 以 自 动 检 测 刻 板 类 并 在 ApplicationContext 中 注 册 对 应 的 BeanDefinition。比如，下面的两个类就是自动检测的例子：

```
@Service
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
@Repository
public class JpaMovieFinder implements MovieFinder {
    //为了清晰省略了实现
}
```

要自动检测这些类并注册对应的 bean ，你需要包含如下的 XML 元素 ，其中的 base-package 元素通常是这两个类的的父包。（也就是说，你可以指定以逗号分隔的列表来 为每个类引入父包。）

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns
:xsi="http://www.w3.org/2001/XMLSchema -instance" xmlns:context=
"http://www.springframework.org/schema/context" xsi:schemaLocati
on="http://www.springframework.org/schema/beans http://www.sprin
gframework.org/schema/beans/spring -beans-3.0.xsd http://www.spr
ingframework.org/schema/context http://www.springframework.org/s
chema/context/spring-context-3.0.xsd">
    <context:component-scan base-package="org.example"/>
</beans>
```

> 注意
>
> 对类路径包的扫描需要存在类路径下对应的目录。当你使用 Ant 来构建 JAR 包时，要保 证你没有激活 JAR 目标中的 files-only 开关。

此外，当你使用 component-scan（组件-扫描，译者注）时，AutowiredAnnotationBeanPostProcessor 和 CommonAnnotationBeanPostProcessor 二者都是隐式包含的。这就意味着两个组件 被自动检测之后就装配在一起了-而不需要在 XML 中提供其它任何 bean 的配置元数据。

> 注意
>
> 你可以将 annotation-config 属性置为 false 来关闭 AutowiredAnnotationBeanPostProcessor 的 CommonAnnotationBeanPostProcessor 注册。

4.10.3 使用过滤器来自定义扫描

默认情况下，使用@Component，@Repository，@Service，@Controller 注解 或使用了进行自定义的@Component 注解的类本身仅仅检测候选组件。你可以修改并扩展 这 种 行 为， 仅仅 应 用自 定 义的 过 滤器 就 可以 了。 在 component-scan 元素中添加 include-filter 或 exclude-filter 子元素就可以了。每个过滤器元素需要 type 和 expression 属性。下面的表格描述了过滤选项。

表 4.5 过滤器类型

过滤器类型	表达式示例	描述
annotation（注解）	org.example.SomeAnnotation	在目标组件的类型层表示的注
assignable（分配）	org.example.SomeClass	目标组件分配去（扩展/实现）口）
aspectj	org.example..*Service+	AspectJ 类型表达式来匹配目
regex（正则表达式）	org.example.Default.*	正则表达式来匹配目标组件类
custom（自定义）	org.example.MyTypeFilter	自 定 义 org.springframework.core.typ 接口的实现类

下面的示例代码展示了 XML 配置忽略所有@Repository 注解并使用“sub”库来替代。

```
<beans>
    <context:component-scan base-package="org.example">
    <context:include-filter type="regex" expression=".*Stub.*Rep
ository"/>
        <context:exclude-filter type="annotation"
        expression="org.springframework.stereotype.Repository
        "/>
    </context:component-scan>
</beans>
```

> 注意
>
> 你可以使用 `<component-scan/>` 元素中的 use-default-filter="false"属性来关闭默认的过滤 器。这会导致关闭使用@Component，@Repository，@Service 或@Controller 注解 的类的自动检测。

4.10.4 使用组件定义 bean 的元数据

Spring 组件可以为容器提供 bea n 定义的元数据。你可以使用用于定义 bea n 元数据的@Configuration 注解的类的@Bean 注解。这里有一个示例：

```
@Component
public class FactoryMethodComponent {
    @Bean @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }
    public void doWork() {
        //忽略组件方法的实现
    }
}
```

这个类在 Spring 的组件中有包含在它的 doWork()方法中的特定应用代码。它也提供 了 bean 的定义并且有工厂方法来指向 publicInstance()方法。@Bean 注解标识了工厂 方法和其它 bean 定义的属性，比如通过@Qualifier 注解表示的限定符。其它方法级的注 解可以用于特定的是@Scope，@Lazy 和自定义限定符注解。自动装配字段和方法也是支持 的，这在之前讨论过，而且还有对自动装配@Bean 方法的支持：

```
@Component
public class FactoryMethodComponent {
    private static int i;
    @Bean @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }
    // 使用自定义标识符并自动装配方法参数
    @Bean
    protected TestBean protectedInstance(@Qualifier( "public") T
estBean spouse,@Value("#{privateInstance.age}") String country)
{
        TestBean tb = new TestBean("protectedInstance", 1); tb.s
etSpouse(tb);
        tb.setCountry(country);
        return tb;
    }
    @Bean @Scope(BeanDefinition.SCOPE_SINGLETON)
    private TestBean privateInstance() {
        return new TestBean("privateInstance", i++);
    }
    @Bean @Scope(value = WebApplicationContext.SCOPE_SESSION, pr
oxyMode = ScopedProxyMode.TARGET_CLASS)
    public TestBean requestScopedInstance() {
        return new TestBean("requestScopedInstance", 3);
    }
}
```

这个示例为另外一个名为 privateInstance 的 bea n 的 Age 属性自动装配了 String 方法参数 country。Spring 的表达式语言元素通过 `#{<expression>}` 表示定义了属性的 值。对于@Value 注解，当解析表达式文本时，表达式解析器会预先配置来查看 bean 的名 称。

Spring 组件中的@Bean 方法会被不同方式来处理，而不会像 Spring 中@Configuration 的类那样。不同的是@Component 类没有使用 CGLIB 来加强并拦截字段和方法的调用。CGLIB 代理是调用@Configuration 类和创建 bean 元数据引用协作对象的@Bean 方法调用的手 段。方法没有使用通常的 Java 语义来调用。相比之下，使用@Component 类@Bean 方法来 调用方法或字段有标准的 Java 语义。

4.10.5 命名自动检测组件

当组件被自动检测作为扫描进程的一部分时，它的 bean 名称是由 BeanNameGenerator 策略来生成并告知扫描器的。默认情况下， Spring 的刻板注解（@Component，@Repository，@Service 和@Controller）包含 name 值从而提供对应 bean 定义的名称。 注意 JSR 330 的@Named 注解可以被用于检测组件和为它们提供名称的手段。如果在类路径 中有 JSR 330 的 JAR 包的话，这种行为会被自动开启。

如果注解包含 name 值或者对于其它任意被检测的组件（比如那些被自定义过滤器发现 的），默认的 bean 的名称生成器返回未大写的非限定符类名。比如，如果下面的两个组件被检测到了，那么名称可能是 myMovieLister 和 movieFinderImpl：

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

> 注意
>
> 如果你不想使用默认的 bean 命名策略，你可以提供自定义的 bean 命名策略。首先， 实现 BeanNameGenerator 接口，要保证包含默认的没有参数的构造方法。之后，在配置 扫描器时，要提供类的完全限定名。

```
<beans>
    <context:component-scan base-package="org.example" name-gene
rator="org.example.MyNameGenerator" />
</beans>
```

作为通用的规则，要考虑使用注解指定名称时，其它组件可能会有对它的明确的引用。 另一方面，当容器负责装配时，自动生成名称是可行的。

4.10.6 为自动检测组件提供范围

一般情况下，Spring 管理的组件，自动检测组件的默认和最多使用的范围是单例范围。 然而，有事你需要其它范围，Spring 2.5 提供了一个新的@Scope 注解。仅仅使用注解提供 范围的名称：

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

> 注意
>
> 为范围决策提供自定义策略而不是基于注解的方式，实现 ScopeMetadataResolver 接口，并保证包含了默认的无参构造方法。之后，当配置扫描器时，要提供类的完全限定名：

```
<beans>
    <context:component-scan base-package="org.example" scope-res
olver="org.example.MyScopeResolver" />
</beans>
```

当使用特定反而非单例范围时，它可能必须要为有范围的对象生成代理。这个原因在 4.5.4.5 节，“各种范围的 bean 作为依赖”中描述过了。出于这样的目的，在 component-scan 元素中可以使用 scoped-proxy 属性。三种可能的值是：no（无），interface（接口）和 targetClass（目标类）。比如，下面的配置就会启动标准的 JDK 动态代理：

```
<beans>
    <context:component-scan base-package="org.example" scoped-pr
oxy="interfaces" />
</beans>
```

4.10.7 使用注解提供限定符元数据

@Qualifier 注解在 4.9.3 节，“使用限定符来微调基于注解的自动装配”中讨论过了。 那部分中的示例说明了@Qualifier 注解的使用和当你需要处理自动装配候选者时，自定 义限定符注解来提供微调控制。因为那些示例是基于 XML 的 bean 定义的，限定符元数据在 候选者 bean 定义中提供，并使用了 XML 中的 bean 元素的

qualifier 和 meta 子元素。 当对自动检测组件使用基于类路径扫描时，你可以在候选者类中使用类型-级的注解提供限 定符元数据。下面的三个示例就展示了这个技术：

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
@Component
@Genre("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
@Component
@Offline
public class CachingMovieCatalog implements MovieCatalog {
    // ...
}
```

> 注意
>
> 对于大多数基于注解的方式，要记得注解元数据会绑定到类定义本身中去，而使用 XML 就允许对多个相同类型的 bean 在它们的限定符元数据中提供变化，因为那些元数据是对于 每个实例而不是每个类提供的。

4.11 使用 JSR 330 标准注解

从 Spring 3.0 开始，Spring 提供了对 JSR-330 标准注解（依赖注入）的支持。这些注解可 以和 Spring 注解以相同方式被扫描到。你仅仅需要在类路径下添加相关的 jar 包即可。

> 注意
>
> 如果你使用 Maven，那么在标准 Maven 仓库（http://repo1.maven.org/maven2/javax/inject/javax.inject/1/）中 javax.inject 的 artifact 是可用的。你仅仅需要在 pom.xml 中添加如下的依赖即可：

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

4.11.1 使用@Inject 和@Named 进行依赖注入

除了@Autowired，javax.inject.Inject 还可以是下面这样：

```
import javax.inject.Inject;
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

至于@Autowired，可以在类级别，字段级别，方法级别和构造方法参数级别使用@Inject。如果你想对应该被注入的依赖使用限定符名称，你应该按如下方式使用@Named注解：

```
import javax.inject.Inject; import javax.inject.Named; public cl
ass SimpleMovieLister {
    private MovieFinder movieFinder;
    @Inject
    public void setMovieFinder(@Named("main") MovieFinder movieF
inder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

4.11.2 @Named：@Component 注解的标准等同

除了@Component，javax.inject.Named 还可以是下面这样：

```
import javax.inject.Inject;
import javax.inject.Named;
@Named("movieListener")
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

使用@Component 而不指定组件的名称是很常用的方式。@Named 可以被用于相同的 情况：

```
import javax.inject.Inject;
import javax.inject.Named;
@Named
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
    // ...
}
```

当使用@Named，当使用 Spring 注解时，可以以相同的方式使用组件扫描：

```
<beans>
    <context:component-scan base-package="org.example"/>
</beans>
```

4.11.3 标准方法的限制

当使用标准注解时，了解一些不可用，但是很重要的特性是必要的，在下面的表格中给 出：

表 4.6 Spring 注解和标准注解的对比

Spring	javax.inject.*	javax.inject 限制/注释
@Autowired	@Inject	@Inject 没有'required'属性
@Component	@Named	
@Scope("singleton")	@Singleton	jsr-330 默认范围和 Spring 的 prototype 相似。但是，要保持和 Spring 一般默认值一致，在 Spring 容器中 jsr-330 的 bean 声 明默认是 singleton 的。要使用另外的 范围，你应该使用 Spring 的@Scope 注解。javax.inject 也提供@Scope 注解。不过这仅 仅用于创建你自己的注解。
@Qualifier	@Named	
@Value	-	不等同
@Required	-	不等同
@Lazy	-	不等同

4.12 基于 Java 的容器配置

4.12.1 基本概念：@Configuration 和@Bean

Spring 中新的 Java 配置支持的核心就是@Configuration 注解的类。这些类主要包括 @Bean 注解的方法来为 Spring 的 IoC 容器管理的对象定义实例，配置和初始化逻辑。

使用@Configuration 来注解类表示类可以被 Spring 的 IoC 容器所使用，作为 bean 定义的资源。最简单的@Configuration 类可以是这样的：

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

这和 Spring 的 XML 文件中的 `<beans/>` 非常类似，上面的 AppConfig 类和下面的代码是等同的：

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"
/>
</beans>
```

正如你看到的，@Bean 注解扮演了和 `<bean/>` 元素相同的角色。@Bean 注解会在下面 的章节中详细地讨论。首先，我们要来看看使用基于 Java 的配置创建 Spring 容器的各种方 式。

4.12.2 使用 AnnotationConfigApplicationContext 实例化 Spring 容器

下面的章节说明了 Spring 的 AnnotationConfigApplicationContext，在 Spring 3.0 中 是 新加 入 的 。这 个 全 能的 ApplicationContext 实 现 类 可 以 接 受不 仅 仅是 @Configuration 类作为输入，也可以是普通的@Component 类，还有使用 JSR-330 元数 据注解的类。

当@Configuration 类作为输入时，@Configuration 类本身作为 bean 被注册了，并且类内所有声明的@Bean 方法也被作为 bean 注册了。

当@Component 和 JSR-330 类 作为输 入时， 它们 被注册 为 bea n，并 且被 假设如 @Autowired 或@Inject 的 DI 元数据在类中需要的地方使用。

4.12.2.1 简单构造

当实例化 ClassPathXmlApplicationContext 时，以大致相同的方式，当实例化 AnnotationConfigApplicationContext 时，@Configuration 类可能被作为输入。这就允许在 Spring 容器中完全可以不使用 XML：

```
public static void main(String[] args) {
    ApplicationContext ctx
        = new AnnotationConfigApplicationContext(AppConfig. clas
s);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

正如上面所提到的，AnnotationConfigApplicationContext 不仅仅局限于和 @Configuration 类合作。任意@Component 或 JSR-330 注解的类都可以作为构造方法的输入。比如：

```
public static void main(String[] args) {
    ApplicationContext ctx = new
        AnnotationConfigApplicationContext(MyServiceImpl.class,
    Dependency1.class, Dependency2.MyService myService =
    ctx.getBean(MyService.class);
    myService.doStuff();
}
```

上面假设 MyServiceImpl，Dependency1 和 Dependency2 使用了 Spring 依赖注入注解，比如@Autowired。

4.12.2.2 使用 `register(Class<?>...)` 来编程构建容器

AnnotationConfigApplicationContext 可以使用无参构造方法来实例化，之后使用 register() 方法来配置。这个方法当编程构建 AnnotationConfigApplicationContext 时尤其有用。

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new
        AnnotationConfigApplicationContext();
    ctx.register(AppConfig.class, OtherConfig.class);
    ctx.register(AdditionalConfig.class);
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

4.12.2.3 使用 scan(String..)开启组件扫描

有经验的 Spring 用户肯定会熟悉下面这个 Spring 的 context:命名空间中的常用 XML 声明

```
<beans>
    <context:component-scan base-package="com.acme"/>
</beans>
```

在上面的示例中，com.acme 包就会被扫描，去查找任意@Component 注解的类，那些 类就会被注册为 Spring 容器中的 bean。AnnotationConfigApplicationContext 暴露 出 scan(String ...)方法，允许相同的组件扫描功能：

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new
    AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

> 注意
>
> 记得@Configuration 类是使用@Component 进行元数据注释的，所以它们是组件扫 描的候选者！在上面的示例中，假设 AppConfig 是声明在 com.acme 包（或是其中的子 包）中的，那么会在调用 scan()方法时被找到，在调用 refresh()方法时，所有它的@Bean 方法就会被处理并注册为容器中的 bean。

4.12.2.4 支持 Web 应用的 AnnotationConfigWebApplicationContext

WebApplicationContext 是 AnnotationConfigApplicationContext 的变种，

适用于 AnnotationConfigWebApplicationContext。当配置 Spring 的 Servlet 监听器 ContextLoaderListener，Spring MVC 的 DispatcherServlet 等时，这个实现类就 可能被用到了。下面的代码是在 web.xml 中的片段，配置了典型的 Spring MVC 的 Web 应用 程序。注意 contextClass 上下文参数和初始化参数的使用：

```
<web-app>
    <!-- 配置ContextLoaderListener使用
    AnnotationConfigWebApplicationContext来代替默认的 XmlWebApplic
ationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value> org.springframework.web.context.support.An
```

```
notationCon
        figWebApplicationContext
        </param-value>
    </context-param>
    <!-- 配置位置必须包含一个或多个逗号或空格分隔的完全限定
    @Configuration类。完全限定包也可以指定于组件扫描 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.AppConfig</param-value>
    </context-param>
    <!-- 普通方式启动根应用上下文的ContextLoaderListener -->
    <listener>
        <listener-class> org.springframework.web.context.Context
LoaderListener
        </listener-class>
    </listener>
    <!-- 普通方式声明 Spring MVC 的 DispatcherServlet -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
            <servlet-class> org.springframework.web.servlet.Disp
atcherServlet
        </servlet-class>
        <!-- 配置DispatcherServlet使用
        AnnotationConfigWebApplicationContext来代替默认的 XmlWebAp
plicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value> org.springframework.web.context.suppor
t.Annotation
                ConfigWebApplicationContext
            </param-value>
        </init-param>
        <!--配置位置必须包含一个或多个逗号或空格分隔的完全限定
        @Configuration类-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.acme.web.MvcConfig</param-value>
        </init-param>
    </servlet>
    <!-- 映射所有的请求到/main/*中来派发servlet -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/main/*</url-pattern>
    </servlet-mapping>
</web-app>
```

4.12.3 构成基于 Java 的配置

4.12.3.1 使用@Import 注解

就像 Spring 的 XML 文件中使用的 `<import/>` 元素帮助模块化配置一样，@Import 注 解允许从其它配置类中加载@Bean 的配置：

```
@Configuration
public class ConfigA {
    public @Bean A a() { return new A(); }
}
@Configuration
@Import(ConfigA.class)
public class ConfigB {
    public @Bean B b() { return new B(); }
}
```

现在，当实例化上下文时，不需要指定 ConfigA.class 和 ConfigB.class 了，仅 仅 ConfigB 需要被显式提供：

```
public static void main(String[] args) {
    ApplicationContext ctx = new
        AnnotationConfigApplicationContext(ConfigB.class);
    // 现在bean A 和bean B都会是可用的...
    A a = ctx.getBean(A.class);
    B b = ctx.getBean(B.class);
}
```

这种方式简化了容器的实例化，仅仅是一个类需要被处理，而不是需要开发人员在构造 时记住很多大量的@Configuration 类。

在引入的@Bean 定义中注入依赖 上面的示例是可行的，但是很简单。在很多实际场景中，bean 会有依赖其它配置的类的依赖。当使用 XML 时，这本身不是什么问题，因为没有调用编译器，而且我们可以仅仅 声明 ref="someBean" 并且相信 Spring 在 容 器 初 始化 时 可以 完 成。 当 然 ，当 使 用 @Configuration 的类时，Java 编译器在配置模型上放置约束，对其它 bean 的引用必须 是符合 Java 语法的。

幸运的是，解决这个问题分层简单。记得@Configuration 类最终是容器中的 bean-这就是说它们可以像其它 bean 那样利用@Autowired 注入元数据！ 我们来看一个更加真实的语义，有几个@Configuration 类，每个都依赖声明在其它类中的 bean：

```
@Configuration
public class ServiceConfig {
    private @Autowired AccountRepository accountRepository;
    public @Bean TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }
}
@Configuration
public class RepositoryConfig {
    private @Autowired DataSource dataSource;
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
}
@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {
    public @Bean DataSource dataSource() { /* 返回新的数据源 */ }
}
public static void main(String[] args) {
    ApplicationContext ctx = new
        AnnotationConfigApplicationContext(SystemTestConfig.clas
s);
    // 所有的装配都使用配置的类...
    TransferService transferService = ctx.getBean(TransferServic
e.class); transferService.transfer(100.00, "A123", "C456");
}
```

完全限定引入的 bean 便于导航

在上面的场景中，使用@Autowired 工作正常，提供所需的模块化，但是准确地决定 在哪儿声明自动装配的 bean 还是有些含糊。比如，作为开发者来看待 ServiceConfig， 你如何准确知道@Autowired AccountRepository 在哪里声明的？它没有显式地出现 在代码中，这可能很不错。要记得 SpringSource Tool Suite 提供工具可以生成展示所有对象是 如何装配起来的图片-那可能就是你所需要的。而且，你的 Java IDE 可以很容器发现所有的声明，还有使用的 AccountRepository 类型，也会很快地给你展示出@Bean 方法的位置 和返回的类型。

在这种歧义不被接受和你想有直接从 IDE 中从一个@Configuration 类到另一个导航 的情景中，要考虑自动装配配置类的本身：

```
@Configuration
public class ServiceConfig {
    private @Autowired RepositoryConfig repositoryConfig;
    public @Bean TransferService transferService() {
        // 通过配置类到@Bean方法的导航!
        return new
            TransferServiceImpl(repositoryConfig.accountReposito
ry());
    }
}
```

在上面的情形中，定义 AccountRepository 是完全明确的。而 ServiceConfig 却紧紧耦合到 RepositoryConfig 中了；这就需要我们来权衡了。这种紧耦合可以使用基于接口或抽象基类的@Configuration 类来减轻。考虑下面的代码：

```
@Configuration
public class ServiceConfig {
private @Autowired RepositoryConfig repositoryConfig;
public @Bean TransferService transferService() {
    return new
        TransferServiceImpl(repositoryConfig.accountRepository()
);
}
}
@Configuration
public interface RepositoryConfig {
    @Bean AccountRepository accountRepository();
}
@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig
{
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(...);
    }
}
@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class})
// 导入具体的配置!
public class SystemTestConfig {
    public @Bean DataSource dataSource() { /* 返回数据源 */ }
}
public static void main(String[] args) {
    ApplicationContext ctx
        = new AnnotationConfigApplicationContext(SystemTestConfi
g.class);
    TransferService transferService = ctx.getBean(TransferServic
e.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

现在 ServiceConfig 和 DefaultRepositoryConfig 的耦合就比较松了，并且内 建的 IDE 工具也一直有效：对于开发人员来说会更加简单地获取 RepositoryConfig 实现类的类型层次。以这种方式，导航@Configuration 类和它们的依赖就和普通的基于接口 代码的导航过程没有任何区别了。

4.12.3.2 结合 Java 和 XML 配置

Spring 的@Configuration 类并不是完全 100%地支持 Spring XML 替换的。一些基本特 性，比如 Spring XML 的命名空间会保持在一个理想的方式下去配置容器。在 XML 便于使用 或是必须要使用的情况下，你也有另外一个选择：以“XML 为中心”的方式来实例化容器， 比如，ClassPathXmlApplicationContext，或者以“ Java 为中心”的方式，使用 AnnotationConfigurationApplicationContext 和 @ImportResource 注解来引 入需要的 XML。

以“XML 为中心”使用@Configuration 类从一种特定的方式的包含@Configuration 类的 XML 文件启动 Spring 容器是不错的。 比如，在使用了 Spring XML 的大型的代码库中，根据需要并从已有的 XML 文件中创建 @Configuration 类是很简单的。在下面，你会发现在这种“XML 为中心”情形下，使用 @Configuration 类的选择。

以普通的 Spring `<bean/>` 元素声明@Configuration 类

要记得@Configuration 的类最终仅仅是容器中的 bean。在这个示例中，我们创建了 名为 AppConfig 的@Configuration 类，并且将它包含在 system-test-config.xml 文件中作为 `<bean/>` 的定义。因为开启了 `<context:annotation-config/>` 配置，容器 会识别@Configuration 注解，以合适的方式处理声明在 AppConfig 中 @Bean 方法。

```
@Configuration
public class AppConfig {
    private @Autowired DataSource dataSource;
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
    public @Bean TransferService transferService() {
        return new TransferService(accountRepository());
    }
}

system-test-config.xml
<beans>
    <!-- 开启处理注解功能，比如@Autowired和@Configuration -->
    <context:annotation-config/>
    <context:property-placeholder
        location="classpath:/com/acme/jdbc.properties"/>
    <bean class="com.acme.AppConfig"/>
    <bean class="org.springframework.jdbc.datasource.DriverManag
erData Source">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>

jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=

public static void main(String[] args) {
    ApplicationContext ctx
        = new ClassPathXmlApplicationContext("classpath:/com/acm
e/system-t est-config.xml");
    TransferService transferService = ctx.getBean(TransferServic
e.class);
    // ...
}
```

> 注意
>
> 在上面的 system-test-config.xml 文件中，AppConfig 的 `<bean/>` 定义没有声明 id 元素。这么做也是可以接受的，就不必让其它 bean 去引用它了。同时也就不可能从 容器中通过明确的名称来获取它了。同样地，DataSource bean，通过类型自动装配，那么明确的 bean id 就不严格要求了。

使用 `<context:component-scan/>` 来检索@Configuration 类

因为@Configuration 是使用@Component 来元数据注解的，被@Configuration 注解的类是自动作为组件扫描的候选者的。使用上面相同的语义，我们可以重新来定义 system-test-config.xml 文件来利用组件扫描的优点。注意这种情况下，我们不需要 明确地声明 `<context:annotation-config/>` ，因为 `<context:component-scan/>` 开启了相同的功能。

```
system-test-config.xml
<beans>
    <!-- 选择并注册AppConfig作为bean -->
    <context:component-scan base-package="com.acme"/>
    <context:property-placeholder
        location="classpath:/com/acme/jdbc.properties"/>
    <bean class="org.springframework.jdbc.datasource.DriverManag
erData Source">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

使用了@ImportResource 导入 XML 的@Configuration 类为中心在@Configuration 类作为配置容器主要机制的应用程序中，使用一些 XML 还是必要的。 在这些情况中，仅仅使用@ImportResource 来定义 XML 就可以了。这么来做就实现了“Java 为中心”的方式来配置容器并保持 XML 在最低限度。

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {
    private @Value("${jdbc.url}") String url;
    private @Value("${jdbc.username}") String username;
    private @Value("${jdbc.password}") String password;
    public @Bean DataSource dataSource() {
        return new DriverManagerDataSource(url, username, passwo
rd);
    }
}

properties-config.xml
<beans>
    <context:property-placeholder
    location="classpath:/com/acme/jdbc.properties"/>
</beans>

jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=

public static void main(String[] args) {
    ApplicationContext ctx
        = new AnnotationConfigApplicationContext(AppConfig. clas
s);
    TransferService transferService = ctx.getBean(TransferServic
e.class);
    // ...
}
```

Hibernate 中文文档 3.2

前言

目录

WARNING! This is a translated version of the English Hibernate reference documentation. The translated version might not be up to date! However, the differences should only be very minor. Consult the English reference documentation if you are missing information or encounter a translation error. If you like to contribute to a particular translation, contact us on the Hibernate developer mailing list.

Translator(s): RedSaga Translate Team 满江红翻译团队 <caoxg@yahoo.com>

在今日的企业环境中，把面向对象的软件和关系数据库一起使用可能是相当麻烦、浪费时间的。Hibernate是一个面向Java环境的对象/关系数据库映射工具。对象/关系数据库映射(object/relational mapping (ORM))这个术语表示一种技术，用来把对象模型表示的对象映射到基于SQL的关系模型数据结构中去。

Hibernate不仅仅管理Java类到数据库表的映射（包括Java数据类型到SQL数据类型的映射），还提供数据查询和获取数据的方法，可以大幅度减少开发时人工使用SQL和JDBC处理数据的时间。

Hibernate的目标是对于开发者通常的数据持久化相关的编程任务，解放其中的95%。对于以数据为中心的程序来说,它们往往只在数据库中使用存储过程来实现商业逻辑,Hibernate可能不是最好的解决方案;对于那些在基于Java的中间层应用中，它们实现面向对象的业务模型和商业逻辑的应用，Hibernate是最有用的。不管怎样，Hibernate一定可以帮助你消除或者包装那些针对特定厂商的SQL代码，并且帮你把结果集从表格式的表示形式转换到一系列的对象去。

如果你对Hibernate和对象/关系数据库映射还是个新手，或者甚至对Java也不熟悉，请按照下面的步骤来学习。

1. 阅读第 1 章 *Hibernate入门* ，这是一篇包含详细的逐步指导的指南。本指南的源代码包含在发行包中，你可以在 `doc/reference/tutorial/` 目录下找到。
2. 阅读第 2 章 *体系结构(Architecture)*来理解Hibernate可以使用的环境。
3. 查看Hibernate发行包中的 `eg/` 目录，里面有一个简单的独立运行的程序。把你的JDBC驱动拷贝到 `lib/` 目录下，修改一下 `src/hibernate.properties` ,指定其中你的数据库的信息。进入命令行，切换到你的发行包的目录，输入 `ant eg` (使用了Ant），或者在Windows操作系统中使用 `build eg` 。
4. 把这份参考文档作为你学习的主要信息来源。
5. 在Hibernate 的网站上可以找到经常提问的问题与解答(FAQ)。

6. 在Hibernate网站上还有第三方的演示、示例和教程的链接。
7. Hibernate网站的“社区(Community Area)”是讨论关于设计模式以及很多整合方案(Tomcat, JBoss AS, Struts, EJB,等等)的好地方。

如果你有问题，请使用Hibernate网站上链接的用户论坛。我们也提供一个JIRA问题追踪系统，来搜集bug报告和新功能请求。如果你对开发Hibernate有兴趣，请加入开发者的邮件列表。（Hibernate网站上的用户论坛有一个中文版面，JavaEye也有Hibernate中文版面,您可以在那里交流问题与经验。）

商业开发、产品支持和Hibernate培训可以通过JBoss Inc.获得。（请查阅：http://www.hibernate.org/SupportTraining/）。 Hibernate是一个专业的开放源代码项目(Professional Open Source project)，也是JBoss Enterprise Middleware System(JEMS),JBoss企业级中间件系统的一个核心组件。

1. 翻译说明

本文档的翻译是在网络上协作进行的，也会不断根据Hibernate的升级进行更新。提供此文档的目的是为了减缓学习Hibernate的坡度，而非代替原文档。我们建议所有有能力的读者都直接阅读英文原文。若您对翻译有异议，或发现翻译错误，敬请不吝赐教，报告到如下email地址：cao at redsaga.com

Hibernate版本3的翻译由满江红翻译团队(RedSaga Translate Team)集体进行，这也是一次大规模网络翻译的试验。在不到20天的时间内，我们完成了两百多页文档的翻译，这一成果是通过十几位网友集体努力完成的。通过这次翻译，我们也有了一套完整的流程，从初译、技术审核一直到文字审核、发布。我们的翻译团队还会继续完善我们的翻译流程，并翻译其他优秀的Java开源资料，敬请期待。

表 **1. Hibernate v3**翻译团队

序号	标题	中文标题	**v3**翻译	**v3**审校	**v3.1**
--	Quickstart with Tomcat	在Tomcat中快速上手(3.1版本中取消）	曹晓钢	zoujm	--
#1	Turtotial	Hibernate入门	Zheng Shuai	-	Sean
#2	Architecture	体系结构	Hilton(BJUG)	厌倦发呆	Sean
#3	Configuration	配置	Goncha	mochow	zcgly
#4	Persistent Classes	持久化类	曹晓钢	mochow	Digita
#5	Basic O/R Mapping	对象/关系数据库映射基础(上)	moxie	Kingfish	张成钅
		对象/关系数据库映射基础(下)	inter_dudu	刘国雄(vincent)	张成钅
#6	Collection Mapping	集合类映射	曹晓钢	robbin	--
#7	Association Mappings	关联关系映射	Robbin	devils.advocate	--

#8	Component Mapping	组件映射	曹晓钢	Robbin	Song qiang
#9	Inheritance Mappings	继承映射	morning(BJUG)	mochow	Liang cheng
#10	Working with objects	与对象共事	程广楠	厌倦发呆	--
#11	Transactions And Concurrency	事务和并发	Robbin	mochow	--
#12	Interceptors and events	继承映射	七彩狼(BJUG)	厌倦发呆	--
#13	Batch processing	批量处理	Kingfish(BJUG)	厌倦发呆	--
#14	HQL: The Hibernate Query Language	HQL: Hibernate查询语言	郑浩(BJUG)	Zheng Shuai	--
#15	Criteria Queries	条件查询	nemo(BJUG)	Zheng Shuai	--
#16	Native SQL	Native SQL查询	似水流年	zoujm	--
#17	Filters	过滤数据	冰云(BJUG)	Goncha	--
#18	XML Mapping	XML映射	edward(BJUG)	Goncha	huxb
#19	Improving performance	性能提升	Wangjinfeng	Robbin	--
#20	Toolset Guide	工具箱指南	曹晓钢	Robbin	--
#21	Example: Parent/Child	示例：父子关系	曹晓钢	devils.advocate	--
#22	Example: Weblog Application	示例：Weblog应用程序	曹晓钢	devils.advocate	--
#23	Example: Various Mappings	示例：多种映射	shidu(BJUG)	冰云	--
#24	Best Practices	最佳实践	曹晓钢	冰云	--

v3.2版本在2006年11月份由曹晓钢更新。

关于我们

满江红.开源, http://www.redsaga.com

从成立之初就致力于Java开放源代码在中国的传播与发展,与国内多个Java团体及出版社有深入交流。坚持少说多做的原则，目前有两个团队，“OpenDoc团队”与“翻译团队”，本翻译文档即为翻译团队作品。OpenDoc团队已经推出包括Hibernate、iBatis、Spring、WebWork的多份开放文档，并于2005年5月在Hibernate开放文档基础上扩充成书，出版了原创书籍：《深入浅出Hibernate》，本书400余页，适合各个层次的Hibernate用户。(http://www.redsaga.com/hibernate_book.html)敬请支持。

北京Java用户组, http://www.bjug.org

Beiing Java User Group，民间技术交流组织，成立于2004年6月。以交流与共享为宗旨，每两周举行一次技术聚会活动。BJUG的目标是，通过小部分人的努力，形成一个技术社群，创建良好的交流氛围，并将新的技术和思想推广到整个IT界，让我们共同进步。

Java视线, http://www.javaeye.com

Java视线在是Hibernate中文论坛（ http://www.hibernate.org.cn ，Hibernate中文论坛是中国最早的Hibernate专业用户论坛，为Hibernate在中国的推广做出了巨大的贡献）基础上发展起来的Java深度技术网站，目标是成为一个高品质的，有思想深度的、原创精神的Java技术交流网站，为软件从业人员提供一个自由的交流技术，交流思想和交流信息的平台。

致谢

还有一些朋友给我们发来了勘误，在此致谢：Kurapica，李毅，李海林。

2. 版权声明

第 1 章 Hibernate入门

目录

1.1. 前言

本章是面向Hibernate初学者的一个入门教程。我们从一个使用驻留内存式(in-memory)数据库的简单命令行应用程序开始, 用易于理解的方式逐步开发。

本章面向Hibernate初学者，但需要Java和SQL知识。它是在Michael Goegl所写的指南的基础上完成的。在这里，我们称第三方库文件是指JDK 1.4和5.0。若使用JDK1.3，你可能需要其它的库文件。

本章的源代码已包含在发布包中，位于 `doc/reference/tutorial/` 目录下。

1.2. 第一部分 － 第一个Hibernate应用程序

首先我们将创建一个简单的基于控制台的(console-based)Hibernate应用程序。由于我们使用Java数据库(HSQL DB)，所以不必安装任何数据库服务器。

假设我们希望有一个小应用程序可以保存我们希望参加的活动（events）和这些活动主办方的相关信息。（译者注：在本教程的后面部分，我们将直接使用event而不是它的中文翻译“活动”，以免混淆。）

我们所做的第一件事就是创建我们的开发目录，并且把所有需要用到的Java库文件放进去。解压缩从Hibernate网站下载的Hibernate发布包，并把 `/lib` 目录下所有需要的库文件拷到我们新建开发目录下的 `/lib` 目录下。看起来就像这样：

```
.
+lib
  antlr.jar
  cglib.jar
  asm.jar
  asm-attrs.jars
  commons-collections.jar
  commons-logging.jar
  ehcache.jar
  hibernate3.jar
  jta.jar
  dom4j.jar
  log4j.jar
```

到编写本文时为止，这些是Hibernate运行所需要的最小库文件集合（注意我们也拷贝了 Hibernate3.jar，这个是最主要的文件）。你正使用的Hibernate版本可能需要比这更多或少一些的库文件。请参见发布包中的 `lib/` 目录下的 `README.txt` ，以获取更多关于所需和可选的第三方库文件信息（事实上，Log4j并不是必须的库文件，但被许多开发者所喜欢）。

接下来我们创建一个类，用来代表那些我们希望储存在数据库里的event。

1.2.1. 第一个class

我们的第一个持久化类是一个带有一些属性（property）的简单JavaBean类：

```
package events;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

你可以看到这个类对属性的存取方法（getter and setter method）使用了标准JavaBean命名约定，同时把类属性（field）的访问级别设成私有的（private）。这是推荐的设计，但并不是必须的。Hibernate也可以直接访问这些field，而使用访问方法（accessor method）的好处是提供了重构时的健壮性（robustness）。为了通过反射机制（Reflection）来实例化这个类的对象，我们需要提供一个无参的构造器（no-argument constructor)。

对一特定的event, `id` 属性持有唯一的标识符（identifier）的值。如果我们希望使用Hibernate提供的所有特性，那么所有的持久化实体（persistent entity）类（这里也包括一些次要依赖类）都需要一个这样的标识符属性。而事实上，大多数应用程序（特别是web应用程序）都需要通过标识符来区别对象，所以你应该考虑使用标识符属性而不是把它当作一种限制。然而，我们通常不会操作对象的标识（identity），因此它的setter方法的访问级别应该声明private。这样当对象被保存的时候，只有Hibernate可以为它分配标识符值。你可看到Hibernate可以直接访问public，private和protected的访问方法和field。所以选择哪种方式完全取决于你，你可以使你的选择与你的应用程序设计相吻合。

所有的持久化类（persistent classes）都要求有无参的构造器，因为Hibernate必须使用Java反射机制来为你创建对象。构造器（constructor）的访问级别可以是private，然而当生成运行时代理（runtime proxy）的时候则要求使用至少是package 级别的访问控制，这样在没有字节码指令（bytecode instrumentation）的情况下，从持久化类里获取数据会更有效率。

把这个Java源代码文件放到开发目录下的 `src` 目录里，注意包位置要正确。 现在这个目录看起来应该像这样：

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
```

下一步，我们把这个持久化类的信息告诉Hibernate。

1.2.2. 映射文件

Hibernate需要知道怎样去加载（load）和存储（store）持久化类的对象。这正是Hibernate映射文件发挥作用的地方。映射文件告诉Hibernate它，应该访问数据库(database)里面的哪个表（table）及应该使用表里面的哪些字段（column）。

一个映射文件的基本结构看起来像这样：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.
dtd">

<hibernate-mapping>
[...]
</hibernate-mapping>
```

注意Hibernate的DTD是非常复杂的。你的编辑器或者IDE里使用它来自动完成那些用来映射的XML元素（element）和属性（attribute）。你也可以在文本编辑器里打开DTD－这是最简单的方式来概览所有的元素和attribute，并查看它们的缺省值以及注释。注意Hibernate不会从web加载DTD文件，但它会首先在应用程序的classpath中查找。DTD文件已包括在 `hibernate3.jar` 里，同时也在Hibernate发布包的 `src/` 目录下。

为缩短代码长度，在以后的例子里我们会省略DTD的声明。当然，在实际的应用程序中，DTD声明是必须的。

在 `hibernate-mapping` 标签（tag）之间, 含有一个 `class` 元素。所有的持久化实体类（再次声明，或许接下来会有依赖类，就是那些次要的实体）都需要一个这样的映射，来把类对象映射到SQL数据库里的表。

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

到目前为止，我们告诉了Hibernate怎样把 `Events` 类的对象持久化到数据库的 `EVENTS` 表里，以及怎样从 `EVENTS` 表加载到 `Events` 类的对象。每个实例对应着数据库表中的一行。现在我们将继续讨论有关唯一标识符属性到数据库表的映射。另外，由于我们不关心怎样处理这个标识符，我们就配置由Hibernate的标识符生成策略来产生代理主键字段。

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>

</hibernate-mapping>
```

`id` 元素是标识符属性的声明， `name="id"` 声明了Java属性的名字 — Hibernate 会使用 `getId()` 和 `setId()` 来访问它。 `column` 属性则告诉Hibernate, 我们使用 `EVENTS` 表的哪个字段作为主键。嵌套的 `generator` 元素指定了标识符生成策略，在这里我们指定 `native` ，它根据已配置的数据库（方言）自动选择最佳的标识符生成策略。Hibernate支持由数据库生成，全局唯一性（globally unique）和应用程序指定（或者你自己为任何已有策略所写的扩展）这些策略来生成标识符。

最后我们在映射文件里面包含需要持久化属性的声明。默认情况下，类里面的属性都被视为非持久化的：

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DAT
E"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

和 `id` 元素一样， `property` 元素的 `name` 属性告诉Hibernate使用哪个getter和setter方法。在此例中，Hibernate会寻找 `getDate()/setDate()` , 以及 `getTitle()/setTitle()` 。

为什么 `date` 属性的映射含有 `column` attribute，而 `title` 却没有？当没有设定 `column` attribute 的时候，Hibernate缺省地使用JavaBean的属性名作为字段名。对于 `title` ，这样工作得很好。然而， `date` 在多数的数据库里，是一个保留关键字，所以我们最好把它映射成一个不同的名字。

另一有趣的事情是 `title` 属性缺少一个 `type` attribute。我们在映射文件里声明并使用的类型，却不是我们期望的那样，是Java数据类型，同时也不是SQL数据库的数据类型。这些类型就是所谓的Hibernate 映射类型（*mapping types*），它们能把Java数据类型转换到SQL数据类型，反之亦然。再次重申，如果在映射文件中没有设置 `type` 属性的话，Hibernate会自己试着去确定正确的转换类型和它的映射类型。在某些情况下这个自动检测机制（在Java 类上使用反射机制）不会产生你所

期待或需要的缺省值。 `date` 属性就是个很好的例子，Hibernate无法知道这个属性（ `java.util.Date` 类型的）应该被映射成：SQL `date` ，或 `timestamp` ，还是 `time` 字段。在此例中，把这个属性映射成 `timestamp` 转换器，这样我们预留了日期和时间的全部信息。

应该把这个映射文件保存为 `Event.hbm.xml` ，且就在 `Event` Java类的源文件目录下。映射文件可随意地命名，但 `hbm.xml` 的后缀已成为Hibernate开发者社区的约定。现在目录结构看起来应该像这样：

```
.
+lib
  <Hibernate and third-party libraries>
+src
 +events
  Event.java
  Event.hbm.xml
```

我们继续进行Hibernate的主要配置。

1.2.3. Hibernate配置

现在我们已经有了一个持久化类和它的映射文件，该是配置Hibernate的时候了。在此之前，我们需要一个数据库。 HSQL DB是种基于Java 的SQL数据库管理系统（DBMS），可以从HSQL DB的网站上下载。实际上，你只需下载的包中的 `hsqldb.jar` 文件，并把这个文件放在开发文件夹的 `lib/` 目录下即可。

在开发的根目录下创建一个 `data` 目录 － 这是HSQL DB存储数据文件的地方。此时在data目录中运行 `java -classpath ../lib/hsqldb.jar org.hsqldb.Server` 就可启动数据库。你可以在log中看到它的启动，及绑定到TCP/IP套结字，这正是我们的应用程序稍后会连接的地方。如果你希望在本例中运行一个全新的数据库，就在窗口中按下 `CTRL + C` 来关闭HSQL数据库，并删除 `data/` 目录下的所有文件，再重新启动HSQL数据库。

Hibernate是你的应用程序里连接数据库的那层，所以它需要连接用的信息。连接（connection）是通过一个也由我们配置的JDBC连接池（connection pool）来完成的。Hibernate的发布包里包含了许多开源的（open source）连接池，但在我们例子中使用Hibernate内置的连接池。注意，如果你希望使用一个产品级(production-quality)的第三方连接池软件，你必须拷贝所需的库文件到你的classpath下，并使用不同的连接池设置。

为了保存Hibernate的配置，我们可以使用一个简单的 `hibernate.properties` 文件，或者一个稍微复杂的 `hibernate.cfg.xml` ，甚至可以完全使用程序来配置Hibernate。多数用户更喜欢使用XML配置文件：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuratio
n-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">org.hsqldb.jdbc
Driver</property>
        <property name="connection.url">jdbc:hsqldb:hsql://local
host</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDiale
ct</property>

        <!-- Enable Hibernate's automatic session context manage
ment -->
        <property name="current_session_context_class">thread</p
roperty>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cach
e.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -
->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource="events/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

注意这个XML配置使用了一个不同的DTD。在这里，我们配置了Hibernate的 `SessionFactory` ——一个关联于特定数据库全局的工厂（factory）。如果你要使用多个数据库，就要用多个的 `<session-factory>` ，通常把它们放在多个配置文件中（为了更容易启动）。

最开始的4个 `property` 元素包含必要的JDBC连接信息。方言（dialect）的 `property` 元素指明Hibernate 生成的特定SQL变量。你很快会看到，Hibernate对持久化上下文的自动session管理就会派上用场。 打开 `hbm2ddl.auto` 选项将自动生成数据库模式（schema）－ 直接加入数据库中。当然这个选项也可以被关闭（通过去除这个配置选项）或者通过Ant任务 `SchemaExport` 的帮助来把数据库schema重定向到文件中。最后，在配置中为持久化类加入映射文件。

把这个文件拷贝到源代码目录下面，这样它就位于classpath的根目录的最后。Hibernate在启动时会自动在classpath的根目录查找名为 `hibernate.cfg.xml` 的配置文件。

1.2.4. 用Ant构建

现在我们用Ant来构建应用程序。你必须先安装Ant－可以从Ant 下载页面得到它。怎样安装Ant就不在这里介绍了，请参考Ant 用户手册。当你安装完了Ant，就可以开始创建 `build.xml` 文件，把它直接放在开发目录下面。

一个简单的build文件看起来像这样：

```
<project name="hibernate-tutorial" default="compile">

    <property name="sourcedir" value="${basedir}/src"/>
    <property name="targetdir" value="${basedir}/bin"/>
    <property name="librarydir" value="${basedir}/lib"/>

    <path id="libraries">
        <fileset dir="${librarydir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${targetdir}"/>
        <mkdir dir="${targetdir}"/>
    </target>

    <target name="compile" depends="clean, copy-resources">
      <javac srcdir="${sourcedir}"
             destdir="${targetdir}"
             classpathref="libraries"/>
    </target>

    <target name="copy-resources">
        <copy todir="${targetdir}">
            <fileset dir="${sourcedir}">
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
    </target>

</project>
```

这将告诉Ant把所有在lib目录下以 `.jar` 结尾的文件拷贝到classpath中以供编译之用。它也把所有的非Java源代码文件，例如配置和Hibernate映射文件，拷贝到目标目录。如果你现在运行Ant，会得到以下输出：

```
C:\hibernateTutorial\>ant
Buildfile: build.xml

copy-resources:
     [copy] Copying 2 files to C:\hibernateTutorial\bin

compile:
    [javac] Compiling 1 source file to C:\hibernateTutorial\bin

BUILD SUCCESSFUL
Total time: 1 second
```

1.2.5. 启动和辅助类

是时候来加载和储存一些 `Event` 对象了，但首先我们得编写一些基础的代码以完成设置。我们必须启动Hibernate，此过程包括创建一个全局的 `SessoinFactory` ，并把它储存在应用程序代码容易访问的地方。 `SessionFactory` 可以创建并打开新的 `Session` 。一个 `Session` 代表一个单线程的单元操作， `SessionFactory` 则是个线程安全的全局对象，只需要被实例化一次。

我们将创建一个 `HibernateUtil` 辅助类（helper class）来负责启动Hibernate和更方便地操作 `SessionFactory` 。让我们来看一下它的实现：

```
package util;

import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure().bui
ldSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be s
wallowed
            System.err.println("Initial SessionFactory creation
failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

这个类不但在它的静态初始化过程（仅当加载这个类的时候被JVM执行一次）中产生全局的 `SessionFactory` ，而且隐藏了它使用了静态singleton的事实。它也可能在应用程序服务器中的JNDI查找 `SessionFactory` 。

如果你在配置文件中给 `SessionFactory` 一个名字，在 `SessionFactory` 创建后，Hibernate会试着把它绑定到JNDI。要完全避免这样的代码，你也可以使用JMX部署，让具有JMX能力的容器来实例化 `HibernateService` 并把它绑定到JNDI。这些高级可选项在后面的章节中会讨论到。

把 `HibernateUtil.java` 放在开发目录的源代码路径下，与放 `events` 的包并列：

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
  +util
    HibernateUtil.java
  hibernate.cfg.xml
+data
build.xml
```

再次编译这个应用程序应该不会有问题。最后我们需要配置一个日志（logging)系统 － Hibernate使用通用日志接口，允许你在Log4j和JDK 1.4 日志之间进行选择。多数开发者更喜欢Log4j：从Hibernate的发布包中（它在 `etc/` 目录下）拷贝 `log4j.properties` 到你的 `src` 目录，与 `hibernate.cfg.xml` .放在一起。看一下配置示例，如果你希望看到更加详细的输出信息，你可以修改配置。默认情况下，只有Hibernate的启动信息才会显示在标准输出上。

示例的基本框架完成了 － 现在我们可以用Hibernate来做些真正的工作。

1.2.6. 加载并存储对象

我们终于可以使用Hibernate来加载和存储对象了，编写一个带有 `main()` 方法的 `EventManager` 类：

```
package events;
import org.hibernate.Session;

import java.util.Date;

import util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate)
 {

        Session session = HibernateUtil.getSessionFactory().getC
urrentSession();

        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        session.save(theEvent);

        session.getTransaction().commit();
    }

}
```

我们创建了个新的 `Event` 对象并把它传递给Hibernate。现在Hibernate负责与SQL打交道，并把 `INSERT` 命令传给数据库。在运行之前，让我们看一下处理 `Session` 和 `Transaction` 的代码。

一个 `Session` 就是个单一的工作单元。我们暂时让事情简单一些，并假设 Hibernate `Session` 和数据库事务是一一对应的。为了让我们的代码从底层的事务系统中脱离出来（此例中是JDBC，但也可能是JTA），我们使用Hibernate `Session` 中的 `Transaction` API。

`sessionFactory.getCurrentSession()` 是干什么的呢？首先，只要你持有 `SessionFactory` （幸亏我们有 `HibernateUtil` ，可以随时获得），大可在任何时候、任何地点调用这个方法。 `getCurrentSession()` 方法总会返回“当前的”工作单元。记得我们在 `hibernate.cfg.xml` 中把这一配置选项调整为"thread"了吗？因此，因此，当前工作单元被绑定到当前执行我们应用程序的Java线程。但是，这并非是完全准确的,你还得考虑工作单元的生命周期范围(scope),它何时开始,又何时结束.

`Session` 在第一次被使用的时候,即第一次调用 `getCurrentSession()` 的时候,其生命周期就开始。然后它被Hibernate绑定到当前线程。当事务结束的时候，不管是提交还是回滚，Hibernate会自动把 `Session` 从当前线程剥离，并且关闭它。假若你再次调用 `getCurrentSession()` ，你会得到一个新的 `Session` ，并且开始一个新的工作单元。这种线程绑定*(thread-bound)*的编程模型（model）是使用Hibernate的最广泛的方式,因为它支持对你的代码灵活分层(事务划分可以和你的数据访问代码分离开来,在本教程的后面部分就会这么做)。

和工作单元的生命周期这个话题相关，Hibernate `Session` 是否被应该用来执行多次数据库操作？上面的例子对每一次操作使用了一个 `Session` ，这完全是巧合，这个例子不是很复杂，无法展示其他方式。Hibernate `Session` 的生命周期可以很灵活，但是你绝不要把你的应用程序设计成为每一次数据库操作都用一个新的Hibernate `Session` 。因此就算下面的例子（它们都很简单）中你可以看到这种用法，记住每次操作一个*session*是一个反模式。在本教程的后面会展示一个真正的(web)程序。

关于事务处理及事务边界界定的详细信息，请参看第 11 章 事务和并发。在上面的例子中，我们也忽略了所有的错误与回滚的处理。

为第一次运行我们的程序，我们得在Ant的build文件中增加一个可以调用得到的target。

```
<target name="run" depends="compile">
    <java fork="true" classname="events.EventManager" classpathr
ef="libraries">
        <classpath path="${targetdir}"/>
        <arg value="${action}"/>
    </java>
</target>
```

`action` 参数（argument）的值是通过命令行调用这个target的时候设置的：

```
C:\hibernateTutorial\>ant run -Daction=store
```

你应该会看到，编译以后，Hibernate根据你的配置启动，并产生一大堆的输出日志。在日志最后你会看到下面这行：

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_I
D) values (?, ?, ?)
```

这是Hibernate执行的 `INSERT` 命令，问号代表JDBC的绑定参数。如果想要看到绑定参数的值或者减少日志的长度，就要调整你在 `log4j.properties` 文件里的设置。

我们想要列出所有已经被存储的events，就要增加一个条件分支选项到main方法中去。

```
if (args[0].equals("store")) {
    mgr.createAndStoreEvent("My Event", new Date());
}
else if (args[0].equals("list")) {
    List events = mgr.listEvents();
    for (int i = 0; i < events.size(); i++) {
        Event theEvent = (Event) events.get(i);
        System.out.println("Event: " + theEvent.getTitle() +
                           " Time: " + theEvent.getDate());
    }
}
```

我们也增加一个新的 `listEvents()` 方法:

```
private List listEvents() {

    Session session = HibernateUtil.getSessionFactory().getCurre
ntSession();

    session.beginTransaction();

    List result = session.createQuery("from Event").list();

    session.getTransaction().commit();

    return result;
}
```

我们在这里是用一个HQL（Hibernate Query Language－Hibernate查询语言）查询语句来从数据库中加载所有存在的 `Event` 对象。Hibernate会生成适当的SQL，把它发送到数据库，并操作从查询得到数据的 `Event` 对象。当然，你可以使用HQL来创建更加复杂的查询。

现在，根据以下步骤来执行并测试以上各项：

- 运行 `ant run -Daction=store` 来保存一些内容到数据库。当然，先得用hbm2ddl来生成数据库schema。
- 现在把 `hibernate.cfg.xml` 文件中hbm2ddl属性注释掉，这样我们就取消了在启动时用hbm2ddl来生成数据库schema。通常只有在不断重复进行单元测试的时候才需要打开它，但再次运行hbm2ddl会把你保存的一切都删掉（*drop*）—— `create` 配置的真实含义是："在创建SessionFactory的时候，从schema 中drop 掉所有的表，再重新创建它们"。

如果你现在使用命令行参数 `-Daction=list` 运行Ant，你会看到那些至今为止我们所储存的events。当然，你也可以多调用几次 `store` 以保存更多的envents。

注意，很多Hibernate新手在这一步会失败，我们不时看到关于*Table not found*错误信息的提问。但是，只要你根据上面描述的步骤来执行，就不会有这个问题，因为hbm2ddl会在第一次运行的时候创建数据库schema，后继的应用程序重起后还能继续使用这个schema。假若你修改了映射，或者修改了数据库schema，你必须把hbm2ddl重新打开一次。

1.3. 第二部分 － 关联映射

我们已经映射了一个持久化实体类到表上。让我们在这个基础上增加一些类之间的关联。首先我们往应用程序里增加人（people）的概念，并存储他们所参与的一个Event列表。（译者注：与Event一样，我们在后面将直接使用person来表示“人”而不是它的中文翻译）

1.3.1. 映射Person类

最初简单的 `Person` 类：

```
package events;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for '
id'

}
```

创建一个名为 `Person.hbm.xml` 的新映射文件（别忘了最上面的DTD引用）：

```
<hibernate-mapping>

    <class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

最后，把新的映射加入到Hibernate的配置中：

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

现在我们在这两个实体之间创建一个关联。显然，persons可以参与一系列events，而events也有不同的参加者（persons）。我们需要处理的设计问题是关联方向（directionality），阶数（multiplicity）和集合（collection）的行为。

1.3.2. 单向Set-based的关联

我们将向 `Person` 类增加一连串的events。那样，通过调用 `aPerson.getEvents()` ，就可以轻松地导航到特定person所参与的events，而不用去执行一个显式的查询。我们使用Java的集合类（collection）： `Set` ，因为set 不包含重复的元素及与我们无关的排序。

我们需要用set 实现一个单向多值关联。让我们在Java类里为这个关联编码，接着映射它：

```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

在映射这个关联之前，先考虑一下此关联的另外一端。很显然，我们可以保持这个关联是单向的。或者，我们可以在 `Event` 里创建另外一个集合，如果希望能够双向地导航，如： `anEvent.getParticipants()` 。从功能的角度来说，这并不是必须的。因为你总可以显式地执行一个查询，以获得某个特定event的所有参与者。这是个在设计时需要做出的选择，完全由你来决定，但此讨论中关于关联的阶数是清楚的：即两端都是“多”值的，我们把它叫做多对多*(many-to-many)*关联。因而，我们使用Hibernate的多对多映射：

```
<class name="events.Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="events.Event"/>
    </set>

</class>
```

Hibernate支持各种各样的集合映射， `<set>` 使用的最为普遍。对于多对多关联（或叫*n:m*实体关系）, 需要一个关联表（association table）。 `表` 里面的每一行代表从person到event的一个关联。表名是由 `set` 元素的 `table` 属性配置的。关联里面的标识符字段名，对于person的一端，是由 `<key>` 元素定义，而event一端的字段名是由 `<many-to-many>` 元素的 `column` 属性定义。你也必须告诉Hibernate集合中对象的类（也就是位于这个集合所代表的关联另外一端的类）。

因而这个映射的数据库schema是：

```
    _____________        __________________
   |             |      |                  |       _____________
   |   EVENTS    |      |   PERSON_EVENT   |      |
|
   |_____________|      |__________________|      |    PERSON
|
   |             |      |                  |      |_____________
|
   | *EVENT_ID   | <--> | *EVENT_ID        |      |
|
   |  EVENT_DATE |      | *PERSON_ID       | <--> | *PERSON_ID
|
   |  TITLE      |      |__________________|      |  AGE
|
   |_____________|                                |  FIRSTNAME
|
                                                  |  LASTNAME
|
                                                  |_____________
|
```

1.3.3. 使关联工作

我们把一些people和events 一起放到 `EventManager` 的新方法中：

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurre
ntSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personI
d);
    Event anEvent = (Event) session.load(Event.class, eventId);

    aPerson.getEvents().add(anEvent);

    session.getTransaction().commit();
}
```

在加载一 `Person` 和 `Event` 后，使用普通的集合方法就可容易地修改我们定义的集合。如你所见，没有显式的 `update()` 或 `save()` ，Hibernate会自动检测到集合已经被修改并需要更新回数据库。这叫做自动脏检查（*automatic dirty checking*），你也可以尝试修改任何对象的name或者date属性，只要他们处于持久化状态，也就是被绑定到某个Hibernate 的 `Session` 上（如：他们刚刚在一个单元操作被加载或者保存），Hibernate监视任何改变并在后台隐式写的方式执行SQL。同步内存状态和数据库的过程，通常只在单元操作结束的时候发生，称此过程为清理缓存（*flushing*）。在我们的代码中，工作单元由数据库事务的提交（或者回滚）来结束——这是由 `CurrentSessionContext` 类的 `thread` 配置选项定义的。

当然，你也可以在不同的单元操作里面加载person和event。或在 `Session` 以外修改不是处在持久化（persistent）状态下的对象（如果该对象以前曾经被持久化，那么我们称这个状态为脱管（*detached*））。你甚至可以在一个集合被脱管时修改它：

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurre
ntSession();
    session.beginTransaction();

    Person aPerson = (Person) session
            .createQuery("select p from Person p left join fetch
 p.events where p.id = :pid")
            .setParameter("pid", personId)
            .uniqueResult(); // Eager fetch the collection so we
 can use it detached

    Event anEvent = (Event) session.load(Event.class, eventId);

    session.getTransaction().commit();

    // End of first unit of work

    aPerson.getEvents().add(anEvent); // aPerson (and its collec
tion) is detached

    // Begin second unit of work

    Session session2 = HibernateUtil.getSessionFactory().getCurr
entSession();
    session2.beginTransaction();

    session2.update(aPerson); // Reattachment of aPerson

    session2.getTransaction().commit();
}
```

对 `update` 的调用使一个脱管对象重新持久化，你可以说它被绑定到一个新的单元操作上，所以在脱管状态下对它所做的任何修改都会被保存到数据库里。这也包括你对这个实体对象的集合所作的任何改动（增加/删除）。

这对我们当前的情形不是很有用，但它是非常重要的概念，你可以把它融入到你自己的应用程序设计中。在 `EventManager` 的main方法中添加一个新的动作，并从命令行运行它来完成我们所做的练习。如果你需要person及event的标识符 — 那就用 `save()` 方法返回它（你可能需要修改前面的一些方法来返回那个标识符）：

```
else if (args[0].equals("addpersontoevent")) {
    Long eventId = mgr.createAndStoreEvent("My Event", new Date(
));
    Long personId = mgr.createAndStorePerson("Foo", "Bar");
    mgr.addPersonToEvent(personId, eventId);
    System.out.println("Added person " + personId + " to event "
 + eventId);
}
```

上面是个关于两个同等重要的实体类间关联的例子。像前面所提到的那样，在特定的模型中也存在其它的类和类型，这些类和类型通常是“次要的”。你已看到过其中的一些，像 `int` 或 `String` 。我们称这些类为值类型（*value type*），它们的实例依赖（*depend*）在某个特定的实体上。这些类型的实例没有它们自己的标识（identity），也不能在实体间被共享（比如，两个person不能引用同一个 `firstname` 对象，即使他们有相同的first name）。当然，值类型并不仅仅在JDK中存在（事实上，在一个Hibernate应用程序中，所有的JDK类都被视为值类型），而且你也可以编写你自己的依赖类，例如 `Address` ， `MonetaryAmount` 。

你也可以设计一个值类型的集合，这在概念上与引用其它实体的集合有很大的不同，但是在Java里面看起来几乎是一样的。

1.3.4. 值类型的集合

我们把一个值类型对象的集合加入 `Person` 实体中。我们希望保存email地址，所以使用 `String` 类型，而且这次的集合类型又是 `Set` ：

```
private Set emailAddresses = new HashSet();

public Set getEmailAddresses() {
    return emailAddresses;
}

public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

这个 `Set` 的映射

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR">
    <key column="PERSON_ID"/>
    <element type="string" column="EMAIL_ADDR"/>
</set>
```

比较这次和此前映射的差别，主要在于 `element` 部分，这次并没有包含对其它实体引用的集合，而是元素类型为 `String` 的集合（在映射中使用小写的名字"string"是向你表明它是一个Hibernate的映射类型或者类型转换器）。和之前一样， `set` 元素的 `table` 属性决定了用于集合的表名。 `key` 元素定义了在集合表中外键的字段名。 `element` 元素的 `column` 属性定义用于实际保存 `String` 值的字段名。

看一下修改后的数据库schema。

```
  _____________       __________________
 |             |     |                  |       _____________
 |   EVENTS    |     |   PERSON_EVENT   |      |             |
     ___________________
 |_____________|     |__________________|      |    PERSON   |
     |                   |
 |             |     |                  |      |_____________|
     | PERSON_EMAIL_ADDR |
 | *EVENT_ID   | <--> | *EVENT_ID       |      |             |
     |___________________|
 |  EVENT_DATE |     | *PERSON_ID       | <--> | *PERSON_ID  |
<--> |  *PERSON_ID       |
 |  TITLE      |     |__________________|      |  AGE        |
     |  *EMAIL_ADDR      |
 |_____________|                               |  FIRSTNAME  |
     |___________________|
                                               |  LASTNAME   |
                                               |_____________|
```

你可以看到集合表的主键实际上是个复合主键，同时使用了2个字段。这也暗示了对于同一个person不能有重复的email地址，这正是Java里面使用Set时候所需要的语义（Set里元素不能重复）。

你现在可以试着把元素加入到这个集合，就像我们在之前关联person和event的那样。其实现的Java代码是相同的：

```
private void addEmailToPerson(Long personId, String emailAddress
) {

    Session session = HibernateUtil.getSessionFactory().getCurre
ntSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personI
d);

    // The getEmailAddresses() might trigger a lazy load of the
collection
    aPerson.getEmailAddresses().add(emailAddress);

    session.getTransaction().commit();
}
```

这次我们没有使用*fetch*查询来初始化集合。因此，调用其getter方法会触发另一附加的select来初始化集合，这样我们才能把元素添加进去。检查SQL log，试着通过预先抓取来优化它。

1.3.5. 双向关联

接下来我们将映射双向关联（bi-directional association）— 在Java里让person和event可以从关联的任何一端访问另一端。当然，数据库schema没有改变，我们仍然需要多对多的阶数。一个关系型数据库要比网络编程语言 更加灵活，所以它并不需要任何像导航方向（navigation direction）的东西 — 数据可以用任何可能的方式进行查看和获取。

首先，把一个参与者（person）的集合加入 `Event` 类中：

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

在 `Event.hbm.xml` 里面也映射这个关联。

```
<set name="participants" table="PERSON_EVENT" inverse="true">
    <key column="EVENT_ID"/>
    <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

如你所见，两个映射文件里都有普通的 `set` 映射。注意在两个映射文件中，互换了 `key` 和 `many-to-many` 的字段名。这里最重要的是 `Event` 映射文件里增加了 `set` 元素的 `inverse="true"` 属性。

这意味着在需要的时候，Hibernate能在关联的另一端 — `Person` 类得到两个实体间关联的信息。这将会极大地帮助你理解双向关联是如何在两个实体间被创建的。

1.3.6. 使双向连起来

首先请记住，Hibernate并不影响通常的Java语义。 在单向关联的例子中，我们是怎样在 `Person` 和 `Event` 之间创建联系的？我们把 `Event` 实例添加到 `Person` 实例内的event引用集合里。因此很显然，如果我们要让这个关联可以双向地工作，我们需要在另外一端做同样的事情 － 把 `Person` 实例加入 `Event` 类内的Person引用集合。这“在关联的两端设置联系”是完全必要的而且你都得这么做。

许多开发人员防御式地编程，创建管理关联的方法来保证正确的设置了关联的两端，比如在 `Person` 里：

```
protected Set getEvents() {
    return events;
}

protected void setEvents(Set events) {
    this.events = events;
}

public void addToEvent(Event event) {
    this.getEvents().add(event);
    event.getParticipants().add(this);
}

public void removeFromEvent(Event event) {
    this.getEvents().remove(event);
    event.getParticipants().remove(this);
}
```

注意现在对于集合的get和set方法的访问级别是protected - 这允许在位于同一个包（package）中的类以及继承自这个类的子类可以访问这些方法，但禁止其他任何人的直接访问，避免了集合内容的混乱。你应尽可能地在另一端也把集合的访问级别设成protected。

`inverse` 映射属性究竟表示什么呢？对于你和Java来说，一个双向关联仅仅是在两端简单地正确设置引用。然而，Hibernate并没有足够的信息去正确地执行 `INSERT` 和 `UPDATE` 语句（以避免违反数据库约束），所以它需要一些帮助来正确的处理双向关联。把关联的一端设置为 `inverse` 将告诉Hibernate忽略关联的这一端，把这端看成是另外一端的一个镜象（*mirror*）。这就是所需的全部信息，Hibernate利用这些信息来处理把一个有向导航模型转移到数据库schema时的所有问题。你只需要记住这个直观的规则：所有的双向关联需要有一端被设置为 `inverse`。在一对多关联中它必须是代表多（many）的那端。而在多对多（many-to-many）关联中，你可以任意选取一端，因为两端之间并没有差别。

让我们把进入一个小型的web应用程序。

1.4. 第三部分 - EventManager web应用程序

Hibernate web应用程序使用 `Session` 和 `Transaction` 的方式几乎和独立应用程序是一样的。但是，有一些常见的模式（pattern）非常有用。现在我们编写一个 `EventManagerServlet` 。这个servlet可以列出数据库中保存的所有的events，还提供一个HTML表单来增加新的events。

1.4.1. 编写基本的servlet

在你的源代码目录的 `events` 包中创建一个新的类：

```
package events;

// Imports

public class EventManagerServlet extends HttpServlet {

    // Servlet code
}
```

我们后面会用到 `dateFormatter` 的工具， 它把 `Date` 对象转换为字符串。只要一个formatter作为servlet的成员就可以了。

这个servlet只处理 HTTP `GET` 请求，因此，我们要实现的是 `doGet()` 方法：

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
        throws ServletException, IOException {

    SimpleDateFormat dateFormatter = new SimpleDateFormat("dd.MM
.yyyy");

    try {
        // Begin unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().beginTransaction();

        // Process request and render page...

        // End unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().commit();

    } catch (Exception ex) {
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().rollback()
;
        throw new ServletException(ex);
    }

}
```

我们称这里应用的模式为每次请求一个session*(session-per-request)*。当有请求到达这个servlet的时候，通过对 `SessionFactory` 的第一次调用，打开一个新的Hibernate `Session` 。然后启动一个数据库事务—所有的数据访问都是在事务中进行，不管是读还是写（我们在应用程序中不使用auto-commit模式）。

不要为每次数据库操作都使用一个新的Hibernate `Session` 。将Hibernate `Session` 的范围设置为整个请求。要用 `getCurrentSession()` ，这样它自动会绑定到当前Java线程。

下一步，对请求的可能动作进行处理，渲染出反馈的HTML。我们很快就会涉及到那部分。

最后，当处理与渲染都结束的时候，这个工作单元就结束了。假若在处理或渲染的时候有任何错误发生，会抛出一个异常，回滚数据库事务。这样， `session-per-request` 模式就完成了。为了避免在每个servlet中都编写事务边界界定的代码，可以考虑写一个servlet 过滤器（filter）来更好地解决。关于这一模式的更多信息，请参阅Hibernate网站和Wiki，这一模式叫做*Open Session in View*—只要你考虑用JSP来渲染你的视图（view），而不是在servlet中，你就会很快用到它。

1.4.2. 处理与渲染

我们来实现处理请求以及渲染页面的工作。

```
// Write HTML header
PrintWriter out = response.getWriter();
out.println("<html><head><title>Event Manager</title></head><bod
y>");

// Handle actions
if ( "store".equals(request.getParameter("action")) ) {

    String eventTitle = request.getParameter("eventTitle");
    String eventDate = request.getParameter("eventDate");

    if ( "".equals(eventTitle) || "".equals(eventDate) ) {
        out.println("<b><i>Please enter event title and date.</i
></b>");
    } else {
        createAndStoreEvent(eventTitle, dateFormatter.parse(even
tDate));
        out.println("<b><i>Added event.</i></b>");
    }
}

// Print page
printEventForm(out);
listEvents(out, dateFormatter);

// Write HTML footer
out.println("</body></html>");
out.flush();
out.close();
```

`listEvents()` 方法使用绑定到当前线程的Hibernate `Session` 来执行查询：

```
private void listEvents(PrintWriter out, SimpleDateFormat dateFo
rmatter) {

    List result = HibernateUtil.getSessionFactory()
                    .getCurrentSession().createCriteria(Event.cl
ass).list();
    if (result.size() > 0) {
        out.println("<h2>Events in database:</h2>");
        out.println("<table border='1'>");
        out.println("<tr>");
        out.println("<th>Event title</th>");
        out.println("<th>Event date</th>");
        out.println("</tr>");
        for (Iterator it = result.iterator(); it.hasNext();) {
            Event event = (Event) it.next();
            out.println("<tr>");
            out.println("<td>" + event.getTitle() + "</td>");
            out.println("<td>" + dateFormatter.format(event.getD
ate()) + "</td>");
            out.println("</tr>");
        }
        out.println("</table>");
    }
}
```

最后，`store` 动作会被导向到 `createAndStoreEvent()` 方法，它也使用当前线程的 `Session`：

```
protected void createAndStoreEvent(String title, Date theDate) {
    Event theEvent = new Event();
    theEvent.setTitle(title);
    theEvent.setDate(theDate);

    HibernateUtil.getSessionFactory()
                    .getCurrentSession().save(theEvent);
}
```

大功告成，这个servlet写完了。Hibernate会在单一的 `Session` 和 `Transaction` 中处理到达的servlet请求。如同在前面的独立应用程序中那样，Hibernate可以自动的把这些对象绑定到当前运行的线程中。这给了你用任何你喜欢的方式来对代码分层及访问 `SessionFactory` 的自由。通常，你会用更加完备的设计，把数据访问代码转移到数据访问对象中(DAO模式）。请参见Hibernate Wiki，那里有更多的例子。

1.4.3. 部署与测试

要发布这个程序，你得把它打成web发布包：WAR文件。把下面的脚本加入到你的 `build.xml` 中：

```
<target name="war" depends="compile">
    <war destfile="hibernate-tutorial.war" webxml="web.xml">
        <lib dir="${librarydir}">
          <exclude name="jsdk*.jar"/>
        </lib>

        <classes dir="${targetdir}"/>
    </war>
</target>
```

这段代码在你的开发目录中创建一个 `hibernate-tutorial.war` 的文件。它把所有的类库和 `web.xml` 描述文件都打包进去，web.xml 文件应该位于你的开发根目录中：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://j
ava.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>events.EventManagerServlet</servlet-class
>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

请注意在你编译和部署web应用程之前，需要一个附加的类库： `jsdk.jar` 。这是Java Servlet开发包，假若你还没有，可以从Sun网站上下载，把它copy到你的lib目录。但是，它仅仅是在编译时需要，不会被打入WAR包。

在你的开发目录中，调用 `ant war` 来构建、打包，然后把 `hibernate-tutorial.war` 文件拷贝到你的tomcat的 `webapps` 目录下。假若你还没安装Tomcat，就去下载一个，按照指南来安装。对此应用的发布，你不需要修改任何Tomcat的配置。

在部署完，启动Tomcat之后，通过 `http://localhost:8080/hibernate-tutorial/eventmanager` 进行访问你的应用，在第一次servlet 请求发生时，请在Tomcat log中确认你看到Hibernate被初始化了（ `HibernateUtil` 的静态初始化器被调用），假若有任何异常抛出，也可以看到详细的输出。

1.5. 总结

本章覆盖了如何编写一个简单独立的Hibernate命令行应用程序及小型的Hibernate web应用程序的基本要素。

如果你已经对Hibernate感到自信，通过开发指南目录，继续浏览你感兴趣的内容－那些会被问到的问题大多是事务处理 (第 11 章 事务和并发)，抓取（fetch）的效率 (第 19 章 提升性能)，或者API的使用 (第 10 章 与对象共事)和查询的特性(第 10.4 节 “查询”)。

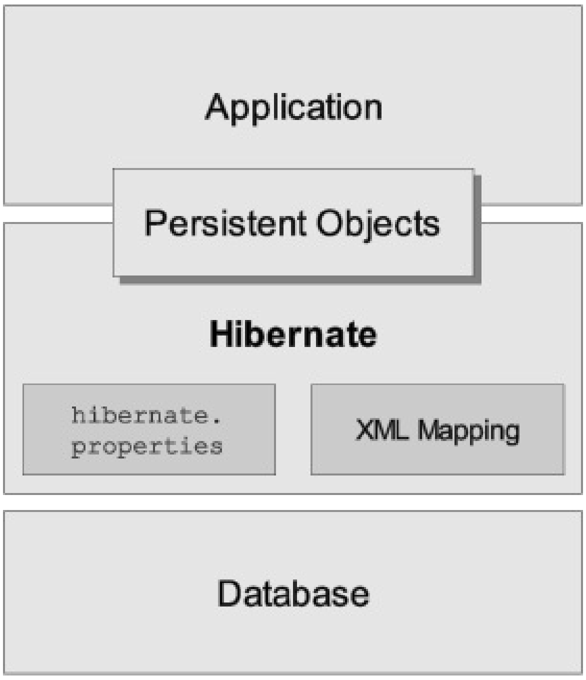
别忘了去Hibernate的网站查看更多（有针对性的）示例。

第 2 章 体系结构(Architecture)

目录

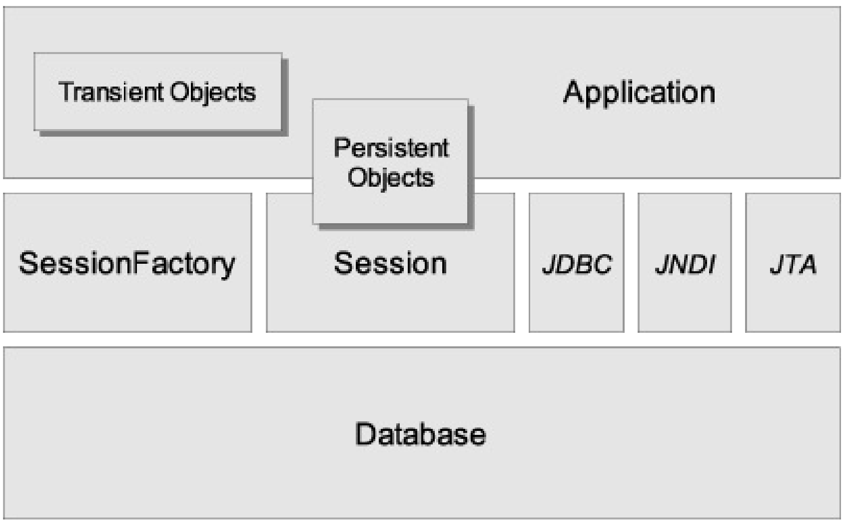
2.1. 概况(Overview)

一个非常简要的Hibernate体系结构的概要图：

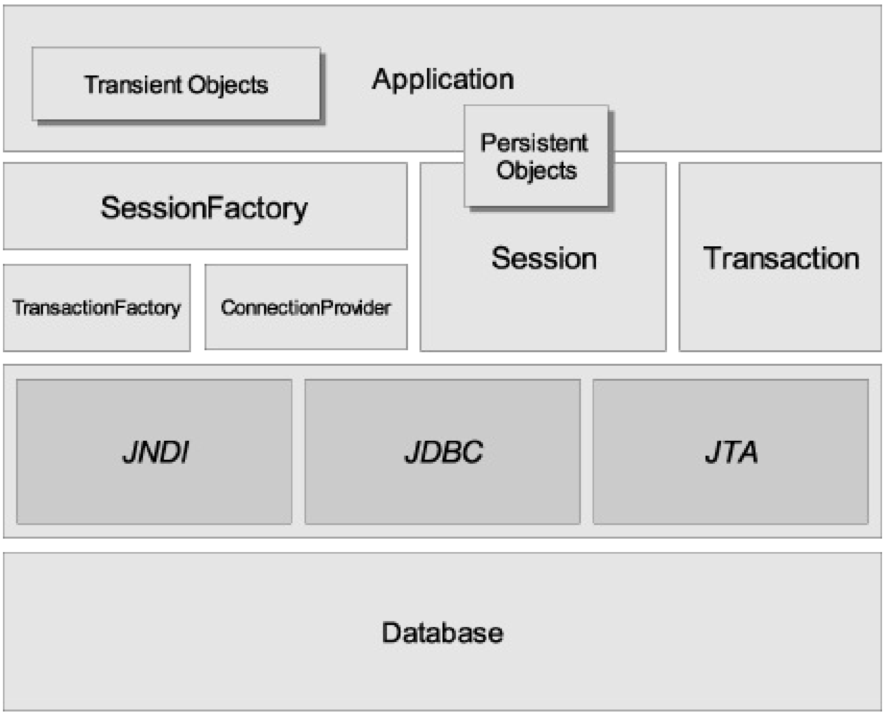


从这个图可以看出，Hibernate使用数据库和配置信息来为应用程序提供持久化服务（以及持久的对象）。

我们来更详细地看一下Hibernate运行时体系结构。由于Hibernate非常灵活，且支持多种应用方案， 所以我们这只描述一下两种极端的情况。“轻型”的体系结构方案，要求应用程序提供自己的JDBC 连接并管理自己的事务。这种方案使用了Hibernate API的最小子集：



“全面解决”的体系结构方案，将应用层从底层的JDBC/JTA API中抽象出来，而让Hibernate来处理这些细节。



图中各个对象的定义如下：

SessionFactory (`org.hibernate.SessionFactory`)

针对单个数据库映射关系经过编译后的内存镜像，是线程安全的（不可变）。 它是生成 `Session` 的工厂，本身要用到 `ConnectionProvider` 。 该对象可以在进程或集群的级别上，为那些事务之间可以重用的数据提供可选的二级缓存。

Session (`org.hibernate.Session`)

表示应用程序与持久储存层之间交互操作的一个单线程对象，此对象生存期很短。其隐藏了JDBC连接，也是 `Transaction` 的工厂。 其会持有一个针对持久化对象的必选（第一级）缓存，在遍历对象图或者根据持久化标识查找对象时会用到。

持久的对象及其集合

带有持久化状态的、具有业务功能的单线程对象，此对象生存期很短。 这些对象可能是普通的JavaBeans/POJO，唯一特殊的是他们正与（仅仅一个） `Session` 相关联。 一旦这个 `Session` 被关闭，这些对象就会脱离持久化状态，这样就可被应用程序的任何层自由使用。 （例如，用作跟表示层打交道的数据传输对象。）

瞬态(transient)和脱管(detached)的对象及其集合

那些目前没有与session关联的持久化类实例。 他们可能是在被应用程序实例化后，尚未进行持久化的对象。 也可能是因为实例化他们的 `Session` 已经被关闭而脱离持久化的对象。

事务Transaction (`org.hibernate.Transaction`)

（可选的）应用程序用来指定原子操作单元范围的对象，它是单线程的，生命周期很短。 它通过抽象将应用从底层具体的JDBC、JTA以及CORBA事务隔离开。 某些情况下，一个 `Session` 之内可能包含多个 `Transaction` 对象。 尽管是否使用该对象是可选的，但无论是使用底层的API还是使用 `Transaction` 对象，事务边界的开启与关闭是必不可少的。

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)

（可选的）生成JDBC连接的工厂（同时也起到连接池的作用）。 它通过抽象将应用从底层的 `Datasource` 或 `DriverManager` 隔离开。 仅供开发者扩展/实现用，并不暴露给应用程序使用。

TransactionFactory (`org.hibernate.TransactionFactory`)

（可选的）生成 `Transaction` 对象实例的工厂。 仅供开发者扩展/实现用，并不暴露给应用程序使用。

扩展接口

Hibernate提供了很多可选的扩展接口，你可以通过实现它们来定制你的持久层的行为。 具体请参考API文档。

在特定“轻型”的体系结构中，应用程序可能绕过 `Transaction` / `TransactionFactory` 以及 `ConnectionProvider` 等API直接跟JTA或JDBC打交道。

2.2. 实例状态

一个持久化类的实例可能处于三种不同状态中的某一种。 这三种状态的定义则与所谓的持久化上下文*(persistence context)*有关。 Hibernate的 `Session` 对象就是这个所谓的持久化上下文：

瞬态（transient）

该实例从未与任何持久化上下文关联过。它没有持久化标识（相当于主键值）。

持久化(persistent)

实例目前与某个持久化上下文有关联。 它拥有持久化标识（相当于主键值），并且可能在数据库中有一个对应的行。 对于某一个特定的持久化上下文，Hibernate保证持久化标识与Java标识（其值代表对象在内存中的位置）等价。

脱管(detached)

实例曾经与某个持久化上下文发生过关联，不过那个上下文被关闭了， 或者这个实例是被序列化(serialize)到另外的进程。 它拥有持久化标识，并且在数据库中可能存在一个对应的行。 对于脱管状态的实例，Hibernate不保证任何持久化标识和Java标识的关系。

2.3. JMX整合

JMX是管理Java组件(Java components)的J2EE标准。 Hibernate 可以通过一个JMX标准服务来管理。 在这个发行版本中，我们提供了一个MBean接口的实现,即 `org.hibernate.jmx.HibernateService` 。

想要看如何在JBoss应用服务器上将Hibernate部署为一个JMX服务的例子，您可以参考JBoss用户指南。 我们现在说一下在Jboss应用服务器上，使用JMX来部署Hibernate的好处：

- *Session*管理： Hibernate的 `Session` 对象的生命周期可以 自动跟一个JTA事务边界绑定。这意味着你无需手工开关 `Session` 了, 这项 工作会由JBoss EJB 拦截器来完成。你再也不用担心你的代码中的事务边界了(除非你想利用Hibernate提供可选 的 `Transaction` API来自己写一个便于移植的的持久层)。 你通过调用 `HibernateContext` 来访问 `Session` 。

- *HAR 部署:* 通常情况下，你会使用JBoss的服务部署描述符（在EAR或/和SAR文件中）来部署Hibernate JMX服务。 这种部署方式支持所有常见的Hibernate `SessionFactory` 的配置选项。 不过，你仍需在部署描述符中，列出你所有的映射文件的名字。如果你使用HAR部署方式, JBoss 会自动探测出你的HAR文件中所有的映射文件。

这些选项更多的描述，请参考JBoss 应用程序用户指南。

将Hibernate以部署为JMX服务的另一个好处，是可以查看Hibernate的运行时统计信息。参看 第 3.4.6 节 “ Hibernate的统计(statistics)机制 ”.

2.4. 对JCA的支持

Hibernate也可以被配置为一个JCA连接器（JCA connector）。更多信息请参看网站。 请注意，Hibernate对JCA的支持，仍处于实验性阶段。

2.5. 上下文相关的（Contextual）Session

使用Hibernate的大多数应用程序需要某种形式的“上下文相关的” session，特定的session在整个特定的上下文范围内始终有效。然而，对不同类型的应用程序而言，要为什么是组成这种“上下文”下一个定义通常是困难的；不同的上下文对“当前”这个概念定义了不同的范围。在3.0版本之前，使用Hibernate的程序要么采用自行编写的基于 `ThreadLocal` 的上下文session，要么采用 `HibernateUtil` 这样的辅助类，要么采用第三方框架（比如Spring或Pico)，它们提供了基于代理(proxy)或者基于拦截器(interception)的上下文相关session。

从3.0.1版本开始，Hibernate增加了 `SessionFactory.getCurrentSession()` 方法。一开始，它假定了采用 `JTA` 事务， `JTA` 事务定义了当前session的范围和上下文(scope and context)。Hibernate开发团队坚信，因为有好几个独立的 `JTA TransactionManager` 实现稳定可用，不论是否被部署到一个 `J2EE` 容器中，大多数(假若不是所有的）应用程序都应该采用 `JTA` 事务管理。基于这一点，采用 `JTA` 的上下文相关session可以满足你一切需要。

更好的是，从3.1开始， `SessionFactory.getCurrentSession()` 的后台实现是可拔插的。因此，我们引入了新的扩展接口 (`org.hibernate.context.CurrentSessionContext`)和新的配置参数 (`hibernate.current_session_context_class`)，以便对什么是“当前session”的范围和上下文(scope and context)的定义进行拔插。

请参阅 `org.hibernate.context.CurrentSessionContext` 接口的Javadoc,那里有关于它的契约的详细讨论。它定义了单一的方法， `currentSession()` ，特定的实现用它来负责跟踪当前的上下文session。Hibernate内置了此接口的三种实现。

- `org.hibernate.context.JTASessionContext` - 当前session根据 `JTA` 来跟踪和界定。这和以前的仅支持JTA的方法是完全一样的。详情请参阅Javadoc。

- `org.hibernate.context.ThreadLocalSessionContext` - 当前session通过当前执行的线程来跟踪和界定。详情也请参阅Javadoc。

- `org.hibernate.context.ManagedSessionContext` - 当前session通过当前执行的线程来跟踪和界定。但是，你需要负责使用这个类的静态方法将 `Session` 实例绑定、或者取消绑定，它并不会打开(open)、flush或者关闭(close)任何 `Session` 。

前两种实现都提供了“每数据库事务对应一个session”的编程模型，也称作每次请求一个*session*。Hibernate session的起始和终结由数据库事务的生存来控制。假若你在纯粹的 Java SE之上采用自行编写代码来管理事务,而不使用JTA，建议你使用Hibernate `Transaction` API来把底层事务实现从你的代码中隐藏掉。如果你使用JTA，请使用JTA借口来管理Transaction。如果你在支持CMT的EJB容器中执行代码，事务边界是声明式定义的，你不需要在代码中进行任何事务或session管理操作。请参阅第 11 章 事务和并发一节来阅读更多的内容和示例代码。

`hibernate.current_session_context_class` 配置参数定义了应该采用哪个 `org.hibernate.context.CurrentSessionContext` 实现。注意，为了向下兼容，如果未配置此参数，但是存在 `org.hibernate.transaction.TransactionManagerLookup` 的配置，Hibernate会采用 `org.hibernate.context.JTASessionContext` 。一般而言，此参数的值指明了要使用的实现类的全名，但那三种内置的实现可以使用简写，即"jta"、"thread"和"managed"。

第 3 章 配置

目录

由于Hibernate是为了能在各种不同环境下工作而设计的, 因此存在着大量的配置参数. 幸运的是多数配置参数都 有比较直观的默认值, 并有随Hibernate一同分发的配置样例 `hibernate.properties` (位于 `etc/`)来展示各种配置选项. 所需做的仅仅是将这个样例文件复制到类路径 (classpath)下并做一些自定义的修改.

3.1. 可编程的配置方式

一个 `org.hibernate.cfg.Configuration` 实例代表了一个应用程序中Java类型到SQL数据库映射的完整集合. `Configuration` 被用来构建一个(不可变的(immutable)) `SessionFactory` . 映射定义则由不同的XML映射定义文件编译而来.

你可以直接实例化 `Configuration` 来获取一个实例，并为它指定XML映射定义 文件. 如果映射定 义文件在类路径(classpath)中, 请使用 `addResource()` :

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

一个替代方法（有时是更好的选择）是，指定被映射的类，让Hibernate帮你寻找映射定义文件:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Hibernate将会在类路径(classpath)中寻找名字为 `/org/hibernate/auction/Item.hbm.xml` 和 `/org/hibernate/auction/Bid.hbm.xml` 映射定义文件. 这种方式消除了任何对文件名的硬编码(hardcoded).

`Configuration` 也允许你指定配置属性:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

当然这不是唯一的传递Hibernate配置属性的方式, 其他可选方式还包括:

1. 传一个 `java.util.Properties` 实例给 `Configuration.setProperties()` .
2. 将 `hibernate.properties` 放置在类路径(classpath)的根目录下 (root directory).

3. 通过 `java -Dproperty=value` 来设置系统 (`System`)属性.
4. 在 `hibernate.cfg.xml` 中加入元素 `<property>` (稍后讨论).

如果想尽快体验Hibernate, `hibernate.properties` 是最简单的方式.

`Configuration` 实例被设计成启动期间（startup-time）对象, 一旦 `SessionFactory` 创建完成它就被丢弃了.

3.2. 获得SessionFactory

当所有映射定义被 `Configuration` 解析后，应用程序必须获得一个用于构造 `Session` 实例的工厂. 这个工厂将被应用程序的所有线程共享:

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate允许你的应用程序创建多个 `SessionFactory` 实例. 这对 使用多个数据库的应用来说很有用.

3.3. JDBC连接

通常你希望 `SessionFactory` 来为你创建和缓存(pool)JDBC连接. 如果你采用这种方式, 只需要如下例所示那样，打开一个 `Session` :

```
Session session = sessions.openSession(); // open a new Session
```

一旦你需要进行数据访问时, 就会从连接池(connection pool)获得一个JDBC连接.

为了使这种方式工作起来, 我们需要向Hibernate传递一些JDBC连接的属性. 所有Hibernate属性的名字和语义都在 `org.hibernate.cfg.Environment` 中定义. 我们现在将描述JDBC连接配置中最重要的设置.

如果你设置如下属性，Hibernate将使用 `java.sql.DriverManager` 来获得(和缓存)JDBC连接 :

表 **3.1. Hibernate JDBC**属性

属性名	用途
`hibernate.connection.driver_class`	*jdbc*驱动类
`hibernate.connection.url`	*jdbc URL*
`hibernate.connection.username`	数据库用户
`hibernate.connection.password`	数据库用户密码
`hibernate.connection.pool_size`	连接池容量上限数目

但Hibernate自带的连接池算法相当不成熟. 它只是为了让你快些上手，并不适合用于产品系统或性能测试中。 出于最佳性能和稳定性考虑你应该使用第三方的连接池。只需要用特定连接池的设置替换 `hibernate.connection.pool_size` 即可。这将关闭Hibernate自带的连接池. 例如, 你可能会想用C3P0.

C3P0是一个随Hibernate一同分发的开源的JDBC连接池, 它位于 `lib` 目录下。 如果你设置了 `hibernate.c3p0.*` 相关的属性, Hibernate将使用 `C3P0ConnectionProvider` 来缓存JDBC连接. 如果你更原意使用Proxool, 请参考发 行包中的 `hibernate.properties` 并到Hibernate网站获取更多的信息.

这是一个使用C3P0的 `hibernate.properties` 样例文件:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabas
e
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

为了能在应用程序服务器(application server)中使用Hibernate, 应当总是将Hibernate 配置成从注册在JNDI中的 `Datasource` 处获得连接，你至少需要设置下列属性中的一个:

表 **3.2. Hibernate**数据源属性

属性名	用途
`hibernate.connection.datasource`	数据源*JNDI*名字
`hibernate.jndi.url`	*JNDI*提供者的*URL* (可选)
`hibernate.jndi.class`	*JNDI* *`InitialContextFactory`* 类 (可选)
`hibernate.connection.username`	数据库用户 (可选)
`hibernate.connection.password`	数据库用户密码 (可选)

这是一个使用应用程序服务器提供的JNDI数据源的 `hibernate.properties` 样例文件:

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

从JNDI数据源获得的JDBC连接将自动参与到应用程序服务器中容器管理的事务(container-managed transactions)中去.

任何连接(connection)属性的属性名都要以" `hibernate.connnection` "开头. 例如, 你可能会使用 `hibernate.connection.charSet` 来指定字符集 `charSet` .

通过实现 `org.hibernate.connection.ConnectionProvider` 接口，你可以定义属于 你自己的获得JDBC连接的插件策略。通过设置 `hibernate.connection.provider_class` ，你可以选择一个自定义的实现.

3.4. 可选的配置属性

有大量属性能用来控制Hibernate在运行期的行为. 它们都是可选的, 并拥有适当的默认值.

警告: 其中一些属性是"系统级(system-level)的". 系统级属性只能通过 `java -Dproperty=value` 或 `hibernate.properties` 来设置, 而不能用上面描述的其他方法来设置.

表 3.3. Hibernate配置属性

属性名	用途
`hibernate.dialect`	一个Hibernate `Dialect` 类名允许Hibernate针对特定的关系数据库生成优化的SQL. 取值 `full.classname.of.Dialect`
`hibernate.show_sql`	输出所有SQL语句到控制台. 有一个另外的选择是把 `org.hibernate.SQL` 这个log category设为 `debug` 。 **eg.** `true` | `false`
`hibernate.format_sql`	在log和console中打印出更漂亮的SQL。 取值 `true` | `false`
`hibernate.default_schema`	在生成的SQL中, 将给定的schema/tablespace附加于非全限定名的表名上. 取值 `SCHEMA_NAME`
`hibernate.default_catalog`	在生成的SQL中, 将给定的catalog附加于非全限定名的表名上. 取值 `CATALOG_NAME`
`hibernate.session_factory_name`	`SessionFactory` 创建后，将自动使用这个名字绑定到JNDI中. 取值 `jndi/composite/name`
`hibernate.max_fetch_depth`	为单向关联(一对一, 多对一)的外连接抓取（outer join fetch）树设置最大深度. 值为 `0` 意味着将关闭默认的外连接抓取. 取值 建议在 `0` 到 `3` 之间取值
`hibernate.default_batch_fetch_size`	为Hibernate关联的批量抓取设置默认数量. 取值 建议的取值

	为 4 , 8 ,和 16
`hibernate.default_entity_mode`	为由这个 `SessionFactory` 打开的所有Session指定默认的实体表现模式. 取值 `dynamic-map` , `dom4j` , `pojo`
`hibernate.order_updates`	强制Hibernate按照被更新数据的主键，为SQL更新排序。这么做将减少在高并发系统中事务的死锁。 取值 `true` | `false`
`hibernate.generate_statistics`	如果开启, Hibernate将收集有助于性能调节的统计数据. 取值 `true` | `false`
`hibernate.use_identifer_rollback`	如果开启, 在对象被删除时生成的标识属性将被重设为默认值. 取值 `true` | `false`
`hibernate.use_sql_comments`	如果开启, Hibernate将在SQL中生成有助于调试的注释信息, 默认值为 `false` . 取值 `true` | `false`

表 **3.4. Hibernate JDBC和连接(connection)属性**

属性名	用途
`hibernate.jdbc.fetch_size`	非零值，指定JDBC抓取数量 用 `Statement.setFetchSi`
`hibernate.jdbc.batch_size`	非零值，允许Hibernate使用 建议取 5 到 30 之间的值
`hibernate.jdbc.batch_versioned_data`	如果你想让你的JDBC驱动从 正确的行计数 , 那么将此属性 选项通常是安全的). 同时，H 化的数据使用批量DML. 默认 `true` | `false`
`hibernate.jdbc.factory_class`	选择一个自定义的 `Batcher` 这个配置属性. **eg.** `classna`
`hibernate.jdbc.use_scrollable_resultset`	允许Hibernate使用JDBC2的 用用户提供的JDBC连接时， 则Hibernate会使用连接的元 `false`
	在JDBC读

hibernate.jdbc.use_streams_for_binary	写 binary (二进制) 或 se 的类型时使用流(stream)(系 false
hibernate.jdbc.use_get_generated_keys	在数据插入数据库之后，允 PreparedStatement.getG 取数据库生成的key(键)。需 JRE1.4+, 如果你的数据库驱 识生成器时遇到问题，请将 下将使用连接的元数据来判 true|false
hibernate.connection.provider_class	自定义 ConnectionProvide Hibernate提供JDBC连接. 取 classname.of.Connectio
hibernate.connection.isolation	设置JDBC事务隔离级别. 查 看 java.sql.Connection 义, 但请注意多数数据库都不 值 1, 2, 4, 8
hibernate.connection.autocommit	允许被缓存的JDBC连接开启 (不建议). 取值 true | fa
hibernate.connection.release_mode	指定Hibernate在何时释放JD 到Session被显式关闭或被断 JDBC连接. 对于应用程序服 当使用 after_statement , 后，都会主动的释放连接. 对 用 after_transaction 在 是合理的. auto 将为JTA和 择 after_statement , 为JI 择 after_transaction .

取值 auto (默认) | on_close | after_transaction | after_statement

注意,这些设置仅对通过 SessionFactory.openSession 得到的 Session 起作用。对于通过 SessionFactory.getCurrentSession 得到的 Session ，所配置的 CurrentSessionContext 实现控制这些 Session 的连接释放模式。请参阅第 2.5 节 “上下文相关的（Contextual）Session”。

|| hibernate.connection._<propertyName>_ | 将JDBC属性 propertyName 传递到 DriverManager.getConnection() 中去. || hibernate.jndi._<propertyName>_ | 将属性 propertyName 传递到 JNDI InitialContextFactory 中去. |

表 **3.5. Hibernate**缓存属性

属性名	用途
`hibernate.cache.provider_class`	自定义的 `CacheProvider` 的 名. 取值 `classname.of.CacheProvi`
`hibernate.cache.use_minimal_puts`	以频繁的读操作为代价, 优化 缓存来最小化写操作. 在 Hibernate3中，这个设置对的 缓存非常有用, 对集群缓存的 而言，默认是开启的. 取值 `true|false`
`hibernate.cache.use_query_cache`	允许查询缓存, 个别查询仍然 被设置为可缓存的. 取值 `true|false`
`hibernate.cache.use_second_level_cache`	能用来完全禁止使用二级缓存 那些在类的映射定义中指 定 `<cache>` 的类，会 认开启二级缓存. 取值 `true|false`
`hibernate.cache.query_cache_factory`	自定义实现 `QueryCache` 接 类名, 默认为内建 的 `StandardQueryCache`. 取 `classname.of.QueryCache`
`hibernate.cache.region_prefix`	二级缓存区域名的前缀. 取值 `prefix`
`hibernate.cache.use_structured_entries`	强制Hibernate以更人性化的 将数据存入二级缓存. 取值 `true|false`

表 3.6. Hibernate事务属性

属性名	
`hibernate.transaction.factory_class`	一个 `TransactionF` Hibernate `Transact` 为 `JDBCTransactio` `classname.of.Tra`
`jta.UserTransaction`	一个JNDI名字，被 `JTATransaction` 器获取JTA `UserTra` `jndi/composite/n`
`hibernate.transaction.manager_lookup_class`	一个 `TransactionM` 使用JVM级缓存，或器的时候需要该类. 耳 `classname.of.Tra`
`hibernate.transaction.flush_before_completion`	如果开启, session在(flush)。 现在更好的下文管理。请参见第（Contextual）Sess `false`
`hibernate.transaction.auto_close_session`	如果开启, session在闭。 现在更好的方法管理。请参见第 2.5（Contextual）Sess `false`

表 **3.7.** 其他属性

属性名	
`hibernate.current_session_context_class`	为"当前" `Session` 指定一 情，请参见第 2.5 节 “上下 `jta` | `thread` | `manag`

`hibernate.query.factory_class`	选择HQL解析器的实现. 取
	`org.hibernate.hql.ast`
	`org.hibernate.hql.cla`

`hibernate.query.substitutions`	将Hibernate查询中的符号 名或常量名字). 取值 `hqlLiteral=SQL_LITERA`
`hibernate.hbm2ddl.auto`	在 `SessionFactory` 创建 schema的DDL导出到数据 闭 `SessionFactory` 时， | `update` | `create` | `c`
`hibernate.cglib.use_reflection_optimizer`	开启CGLIB来替代运行时反 时比较有用. 注意即使关闭 能在 `hibernate.cfg.xml`

3.4.1. SQL方言

你应当总是为你的数据库将 `hibernate.dialect` 属性设置成正确的 `org.hibernate.dialect.Dialect` 子类. 如果你指定一种方言, Hibernate将为上面列出的一些属性使用合理的默认值, 为你省去了手工指定它们的功夫.

表 **3.8. Hibernate SQL方言 (`hibernate.dialect`)**

RDBMS	方言
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL	`org.hibernate.dialect.MySQLDialect`
MySQL with InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
MySQL with MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (any version)	`org.hibernate.dialect.OracleDialect`
Oracle 9i/10g	`org.hibernate.dialect.Oracle9Dialect`
Sybase	`org.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

3.4.2. 外连接抓取(Outer Join Fetching)

如果你的数据库支持ANSI, Oracle或Sybase风格的外连接, 外连接抓取通常能通过限制往返数据库次数 (更多的工作交由数据库自己来完成)来提高效率. 外连接抓取允许在单个 `SELECT` SQL语句中， 通过many-to-one, one-to-many, many-to-many和one-to-one关联获取连接对象的整个对象图.

将 `hibernate.max_fetch_depth` 设为 `0` 能在全局 范围内禁止外连接抓取. 设为 `1` 或更高值能启用one-to-one和many-to-oneouter关联的外连接抓取, 它们通过 `fetch="join"` 来映射.

参见第 19.1 节 “抓取策略(Fetching strategies) ”获得更多信息.

3.4.3. 二进制流 (Binary Streams)

Oracle限制那些通过JDBC驱动传输的 `字节` 数组的数目. 如果你希望使用 `二进值 (binary)` 或 `可序列化的 (serializable)` 类型的大对象, 你应该开启 `hibernate.jdbc.use_streams_for_binary` 属性. 这是系统级属性.

3.4.4. 二级缓存与查询缓存

以 `hibernate.cache` 为前缀的属性允许你在Hibernate中，使用进程或群集范围内的二级缓存系统. 参见第 19.2 节 “二级缓存（The Second Level Cache）”获取更多的详情.

3.4.5. 查询语言中的替换

你可以使用 `hibernate.query.substitutions` 在Hibernate中定义新的查询符号. 例如:

```
hibernate.query.substitutions true=1, false=0
```

将导致符号 `true` 和 `false` 在生成的SQL中被翻译成整数常量.

```
hibernate.query.substitutions toLowercase=LOWER
```

将允许你重命名SQL中的 `LOWER` 函数.

3.4.6. Hibernate的统计(statistics)机制

如果你开启 `hibernate.generate_statistics` , 那么当你通过 `SessionFactory.getStatistics()` 调整正在运行的系统时，Hibernate将导出大量有用的数据. Hibernate甚至能被配置成通过JMX导出这些统计信息. 参考 `org.hibernate.stats` 中接口的Javadoc，以获得更多信息.

3.5. 日志

Hibernate使用Apache commons-logging来为各种事件记录日志.

commons-logging将直接输出到Apache Log4j(如果在类路径中包括 `log4j.jar`)或 JDK1.4 logging (如果运行在JDK1.4或以上的环境下). 你可以从 `http://jakarta.apache.org` 下载Log4j. 要使用Log4j，你需要将 `log4j.properties` 文件放置在类路径下, 随Hibernate 一同分发的样例属性文件在 `src/` 目录下.

我们强烈建议你熟悉一下Hibernate的日志消息. 在不失可读性的前提下， 我们做了很多工作，使Hibernate的日志可能地详细. 这是必要的查错利器. 最令人感兴趣的日志分类有如下这些:

表 3.9. Hibernate日志类别

类别	功能
`org.hibernate.SQL`	在所有SQL DML语句被执行时为它们记录日志
`org.hibernate.type`	为所有JDBC参数记录日志
`org.hibernate.tool.hbm2ddl`	在所有SQL DDL语句执行时为它们记录日志
`org.hibernate.pretty`	在session清洗(flush)时，为所有与其关联的实体(最多20个)的状态记录日志
`org.hibernate.cache`	为所有二级缓存的活动记录日志
`org.hibernate.transaction`	为事务相关的活动记录日志
`org.hibernate.jdbc`	为所有JDBC资源的获取记录日志
`org.hibernate.hql.AST`	在解析查询的时候,记录HQL和SQL的AST分析日志
`org.hibernate.secure`	为JAAS认证请求做日志
`org.hibernate`	为任何Hibernate相关信息做日志 (信息量较大, 但对查错非常有帮助)

在使用Hibernate开发应用程序时, 你应当总是为 `org.hibernate.SQL` 开启 `debug` 级别的日志记录,或者开启 `hibernate.show_sql` 属性。

3.6. 实现 `NamingStrategy`

`org.hibernate.cfg.NamingStrategy` 接口允许你为数据库中的对象和schema 元素指定一个“命名标准”.

你可能会提供一些通过Java标识生成数据库标识或将映射定义文件中"逻辑"表/列名处理成"物理"表/列名的规则. 这个特性有助于减少冗长的映射定义文件.

在加入映射定义前，你可以调用 `Configuration.setNamingStrategy()` 指定一个不同的命名策略:

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

`org.hibernate.cfg.ImprovedNamingStrategy` 是一个内建的命名策略, 对 一些应用程序而言，可能是非常有用的起点.

3.7. XML配置文件

另一个配置方法是在 `hibernate.cfg.xml` 文件中指定一套完整的配置. 这个文件可以当成 `hibernate.properties` 的替代。 若两个文件同时存在，它将覆盖前者的属性.

XML配置文件被默认是放在 `CLASSPATH` 的根目录下. 这是一个例子:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.
0.dtd">

<hibernate-configuration>

    <!-- 以/jndi/name绑定到JNDI的SessionFactory实例 -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- 属性 -->
        <property name="connection.datasource">java:/comp/env/jd
bc/MyDB</property>
        <property name="dialect">org.hibernate.dialect.MySQLDial
ect</property>
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTrans
action</property>

        <!-- 映射定义文件 -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- 缓存设置 -->
        <class-cache class="org.hibernate.auction.Item" usage="r
ead-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="re
ad-only"/>
        <collection-cache collection="org.hibernate.auction.Item
.bids" usage="read-write"/>

    </session-factory>

</hibernate-configuration>
```

如你所见, 这个方法优势在于，在配置文件中指出了映射定义文件的名字. 一旦你需要调整Hibernate的缓存， `hibernate.cfg.xml` 也是更方便. 注意，使用 `hibernate.properties` 还是 `hibernate.cfg.xml` 完全是由你来决定, 除了上面提到的XML语法的优势之外, 两者是等价的.

使用XML配置，使得启动Hibernate变的异常简单, 如下所示，一行代码就可以搞定：

```
SessionFactory sf = new Configuration().configure().buildSession
Factory();
```

你可以使用如下代码来添加一个不同的XML配置文件

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

3.8. J2EE应用程序服务器的集成

针对J2EE体系,Hibernate有如下几个集成的方面:

- 容器管理的数据源*(Container-managed datasources)*: Hibernate能使用通过容器管理，并由JNDI提供的JDBC连接. 通常, 特别是当处理多个数据源的分布式事务的时候, 由一个JTA兼容的 `TransactionManager` 和一个 `ResourceManager` 来处理事务管理(CMT, 容器管理的事务). 当然你可以通过编程方式来划分事务边界(BMT, Bean管理的事务). 或者为了代码的可移植性，你也也许会想使用可选的 Hibernate `Transaction` API.
- 自动*JNDI*绑定: Hibernate可以在启动后将 `SessionFactory` 绑定到JNDI.
- *JTA Session*绑定*:* Hibernate `Session` 可以自动绑定到JTA事务作用的范围. 只需简单地从JNDI查找 `SessionFactory` 并获得当前的 `Session` . 当JTA事务完成时, 让Hibernate来处理 `Session` 的清洗(flush)与关闭. 事务的划分可以是声明式的(CMT),也可以是编程式的(BMT/UserTransaction).
- *JMX*部署*:* 如果你使用支持JMX应用程序服务器(如, JBoss AS), 那么你可以选择将Hibernate部署成托管MBean. 这将为你省去一行从 `Configuration` 构建 `SessionFactory` 的启动代码. 容器将启动你的 `HibernateService` , 并完美地处理好服务间的依赖关系 (在Hibernate启动前，数据源必须是可用的，等等).

如果应用程序服务器抛出"connection containment"异常, 根据你的环境，也许该将配置属性 `hibernate.connection.release_mode` 设为 `after_statement` .

3.8.1. 事务策略配置

在你的架构中，Hibernate的 `Session` API是独立于任何事务分界系统的. 如果你让Hibernate通过连接池直接使用JDBC, 你需要调用JDBC API来打开和关闭你的事务. 如果你运行在J2EE应用程序服务器中, 你也许想用Bean管理的事务并在需要的时候调用JTA API和 `UserTransaction` .

为了让你的代码在两种(或其他)环境中可以移植，我们建议使用可选的Hibernate `Transaction` API, 它包装并隐藏了底层系统. 你必须通过设置Hibernate配置属性 `hibernate.transaction.factory_class` 来指定 一个 `Transaction` 实例的工厂类.

有三个标准(内建)的选择:

`org.hibernate.transaction.JDBCTransactionFactory`

委托给数据库(JDBC)事务（默认）

`org.hibernate.transaction.JTATransactionFactory`

如果在上下文环境中存在运行着的事务(如, EJB会话Bean的方法), 则委托给容器管理的事务, 否则，将启动一个新的事务，并使用Bean管理的事务.

`org.hibernate.transaction.CMTTransactionFactory`

委托给容器管理的JTA事务

你也可以定义属于你自己的事务策略 (如, 针对CORBA的事务服务)

Hibernate的一些特性 (比如二级缓存, Contextual Sessions with JTA等等)需要访问在托管环境中的JTA `TransactionManager` . 由于J2EE没有标准化一个单一的机制,Hibernate在应用程序服务器中，你必须指定Hibernate如何获得 `TransactionManager` 的引用:

表 3.10. JTA TransactionManagers

Transaction工厂类
`org.hibernate.transaction.JBossTransactionManagerLookup`
`org.hibernate.transaction.WeblogicTransactionManagerLookup`
`org.hibernate.transaction.WebSphereTransactionManagerLookup`
`org.hibernate.transaction.WebSphereExtendedJTATransactionLookup`
`org.hibernate.transaction.OrionTransactionManagerLookup`
`org.hibernate.transaction.ResinTransactionManagerLookup`
`org.hibernate.transaction.JOTMTransactionManagerLookup`
`org.hibernate.transaction.JOnASTransactionManagerLookup`
`org.hibernate.transaction.JRun4TransactionManagerLookup`
`org.hibernate.transaction.BESTransactionManagerLookup`

3.8.2. JNDI绑定的 `SessionFactory`

与JNDI绑定的Hibernate的 `SessionFactory` 能简化工厂的查询，简化创建新的 `Session` . 需要注意的是这与JNDI绑定 `Datasource` 没有关系，它们只是恰巧用了相同的注册表!

如果你希望将 `SessionFactory` 绑定到一个JNDI的名字空间，用属性 `hibernate.session_factory_name` 指定一个名字(如, `java:hibernate/SessionFactory`). 如果不设置这个属性, `SessionFactory` 将不会被绑定到JNDI中. (在以只读JNDI为默认实现的环境中，这个设置尤其有用，如Tomcat.)

在将 `SessionFactory` 绑定至JNDI时, Hibernate将使用 `hibernate.jndi.url` ，和 `hibernate.jndi.class` 的值来实例化初始环境(initial context). 如果它们没有被指定，将使用默认的 `InitialContext` .

在你调用 `cfg.buildSessionFactory()` 后, Hibernate会自动将 `SessionFactory` 注册到JNDI. 这意味这你至少需要在你应用程序的启动代码(或工具类)中完成这个调用，除非你使用 `HibernateService` 来做JMX部署 (见后面讨论).

假若你使用JNDI `SessionFactory` ,EJB或者任何其它类都可以从JNDI中找到此 `SessionFactory` 。

我们建议，在受管理的环境中，把 `SessionFactory` 绑定到JNDI，在其它情况下，使用一个 `static(静态的)` singleton。为了在你的应用程序代码中隐藏这些细节，我们还建议你用一个helper类把实际查找 `SessionFactory` 的代码隐藏起来，比如 `HibernateUtil.getSessionFactory()` 。注意，这个类也就可以方便地启动Hibernate，参见第一章。

3.8.3. 在JTA环境下使用Current Session context (当前session上下文)管理

在Hibernate中，管理 `Session` 和transaction最好的方法是自动的"当前" `Session` 管理。请参见第 2.5 节 “上下文相关的（Contextual）Session”一节的讨论。使用 `"jta"` session上下文，假若在当前JTA事务中还没有Hibernate `Session` 关联，第一次 `sessionFactory.getCurrentSession()` 调用会启动一个Session,并关联到当前的JTA事务。在 `"jta"` 上下文中调用 `getCurrentSession()` 获得的 `Session` ，会被设置为在transaction关闭的时候自动flush（清洗）、在transaction关闭之后自动关闭，每句语句之后主动释放JDBC连接。这就可以根据JTA事务的生命周期来管理与之关联的 `Session` ，用户代码中就可以不再考虑这些管理。你的代码也可以通过 `UserTransaction` 用编程方式使用JTA，或者(我们建议，为了便于移植代码）使用Hibernate的 `Transaction` API来设置transaction边界。如果你的代码运行在EJB容器中，建议对CMT使用声明式事务声明。

3.8.4. JMX部署

为了将 `SessionFactory` 注册到JNDI中， `cfg.buildSessionFactory()` 这行代码仍需在某处被执行. 你可在一个 `static` 初始化块(像 `HibernateUtil` 中的那样)中执行它或将Hibernate部署为一个托管的服务.

为了部署在一个支持JMX的应用程序服务器上，Hibernate和 `org.hibernate.jmx.HibernateService` 一同分发，如Jboss AS。 实际的部署和配置是由应用程序服务器提供者指定的. 这里是JBoss 4.0.x 的 `jboss-service.xml` 样例:

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFacto
ry">

    <!-- 必须的服务 -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- 将Hibernate服务绑定到JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</a
ttribute>

    <!-- 数据源设置 -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect<
/attribute>

    <!-- 事务集成 -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribu
te>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup<
/attribute>
    <attribute name="FlushBeforeCompletionEnabled">true</attribu
te>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- 抓取选项 -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- 二级缓存 -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute name="CacheProviderClass">org.hibernate.cache.EhC
acheProvider</attribute>
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- 日志 -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- 映射定义文件 -->
    <attribute name="MapResources">auction/Item.hbm.xml,auction/
Category.hbm.xml</attribute>

</mbean>

</server>
```

这个文件是部署在`META-INF`目录下的, 并会被打包到以`.sar` (service archive)为扩展名的JAR文件中. 同时，你需要将Hibernate、它所需要的第三方库、你编译好的持久化类以及你的映射定义文件打包进同一个文档. 你的企业Bean(一般为会话Bean)可能会被打包成它们自己的JAR文件, 但你也许会将EJB JAR文件一同包含进能独立(热)部署的主服务文档. 参考JBoss AS文档以了解更多的JMX服务与EJB部署的信息.

第 4 章 持久化类(Persistent Classes)

目录

在应用程序中，用来实现业务问题实体的（如，在电子商务应用程序中的Customer和Order） 类就是持久化类。不能认为所有的持久化类的实例都是持久的状态——一个实例的状态也可能 是瞬时的或脱管的。

如果这些持久化类遵循一些简单的规则，Hibernate能够工作得更好，这些规则也被称作 简单传统Java对象(POJO:Plain Old Java Object)编程模型。但是这些规则并不是必需的。 实际上，Hibernate3对于你的持久化类几乎不做任何设想。你可以用其他的方法来表达领域模型： 比如，使用 `Map` 实例的树型结构。

4.1. 一个简单的POJO例子

大多数Java程序需要用一个持久化类来表示猫科动物。

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
```

```
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kitten.setMother(this);
    kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

这里要遵循四条主要的规则：

4.1.1. 实现一个默认的（即无参数的）构造方法（constructor）

`Cat` 有一个无参数的构造方法。所有的持久化类都必须有一个 默认的构造方法（可以不是public的），这样的话Hibernate就可以使用 `Constructor.newInstance()` 来实例化它们。 我们强烈建议，在Hibernate中，为了运行期代理的生成，构造方法至少是 包*(package)*内可见的。

4.1.2. 提供一个标识属性（identifier property）（可选）

`Cat` 有一个属性叫做 `id` 。这个属性映射数据库表的主 键字段。这个属性可以叫任何名字，其类型可以是任何的原始类型、原始类型的包装类型、 `java.lang.String` 或者是 `java.util.Date` 。（如果你的遗留数据库表有联合主键，你甚至可以用一个用户自定义的类，该类拥有这些类型 的属性。参见后面的关于联合标识符的章节。）

标识符属性是可选的。可以不用管它，让Hibernate内部来追踪对象的识别。 但是我们并不推荐这样做。

实际上，一些功能只对那些声明了标识符属性的类起作用：

- 托管对象的传播性再连接（级联更新或级联合并）——参阅 + 第 10.11 节 “传播性持久化(transitive persistence)”
- `Session.saveOrUpdate()`
- `Session.merge()`

我们建议你对持久化类声明命名一致的标识属性。我们还建议你使用一 个可以为空（也就是说，不是原始类型）的类型。

4.1.3. 使用非final的类 (可选)

代理（*proxies*）是Hibernate的一个重要的功能，它依赖的条件是，持久 化类或者是非final的，或者是实现了一个所有方法都声明为public的接口。

你可以用Hibernate持久化一个没有实现任何接口的 `final` 类，但是你 不能使用代理来延迟关联加载，这会限制你进行性能优化的选择。

你也应该避免在非final类中声明 `public final` 的方法。如果你想使用一 个有 `public final` 方法的类，你必须通过设置 `lazy="false"` 来明确地禁用代理。

4.1.4. 为持久化字段声明访问器(accessors)和是否可变的标志(mutators)(可选)

`Cat` 为它的所有持久化字段声明了访问方法。很多其他ORM工具直接对 实例变量进行持久化。我们相信，在关系数据库schema和类的内部数据结构之间引入间接层(原文为"非直接"，indirection)会好一些。默认情况下Hibernate持久化JavaBeans风格的属性，认可 `getFoo` ， `isFoo` 和 `setFoo` 这种形式的方法名。 如果需要，你可以对某些特定属性实行直接字段访问。

属性不需要要声明为public的。Hibernate可以持久化一个有 `default` 、 `protected` 或 `private` 的get/set方法对 的属性进行持久化。

TODO：property和proxy包里的用户扩展框架文档。

4.2. 实现继承（Inheritance）

子类也必须遵守第一条和第二条规则。它从超类 `Cat` 继承了标识属性。

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

4.3. 实现 `equals()` 和 `hashCode()`

如果你有如下需求，你必须重载 `equals()` 和 `hashCode()` 方法：

- 想把持久类的实例放入 `Set` 中（当表示多值关联时，推荐这么做）
- 想重用脱管实例

Hibernate保证，仅在特定会话范围内，持久化标识（数据库的行）和Java标识是等价的。因此，一旦 我们混合了从不同会话中获取的实例，如果希望 `Set` 有明确的语义，就必 须实现 `equals()` 和 `hashCode()` 。

实现 `equals()` / `hashCode()` 最显而易见的方法是比较两个对象 标识符的值。如果值相同，则两个对象对应于数据库的同一行，因此它们是相等的（如果都被添加到 `Set` ，则在 `Set` 中只有一个元素）。不幸的是，对生成的标识不能 使用这种方法。Hibernate仅对那些持久化对象赋标识值，一个新创建的实例将不会有任何标识值。此外， 如果一个实例没有被保存(unsaved)，并且它当前正在一个 `Set` 中，保存它将会给这个对象 赋一个标识值。如果 `equals()` 和 `hashCode()` 是基于标识值 实现的，则其哈希码将会改变，这违反了 `Set` 的契约。建议去Hibernate的站点阅读关于这个 问题的全部讨论。注意，这不是Hibernate的问题，而是一般的Java对象标识和Java对象等价的语义问题。

我们建议使用业务键值相等*(Business key equality)*来实现 `equals()` 和 `hashCode()` 。业务键值相等的意思是， `equals()` 方法 仅仅比较形成业务键的属性，它能在现实世界里标识我们的实例（是一个自然的候选码）。

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return
 false;
        if ( !cat.getMother().equals( getMother() ) ) return fal
se;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

注意，业务键不必像数据库的主键那样固定不变（参见第 11.1.3 节 “关注对象标识 (Considering object identity)”）。对业务键而言，不可变或唯一的属性是不错的选择。

4.4. 动态模型(Dynamic models)

注意，以下特性在当前处于试验阶段，将来可能会有变化。

运行期的持久化实体没有必要一定表示为像POJO类或JavaBean对象那样的形式。Hibernate也支持动态模型 （在运行期使用 `Map` 的 `Map` ）和象DOM4J的树模型那样的实体表示。使用这种方法，你不用写持久化类，只写映射文件就行了。

Hibernate默认工作在普通POJO模式。你可以使用配置选项 `default_entity_mode` ， 对特定的 `SessionFactory` ，设置一个默认的实体表示模式。 （参见表 3.3 “ Hibernate配置属性 ”。）

下面是用 `Map` 来表示的例子。首先，在映射文件中，要声明 `entity-name` 来代替一个类名（或作为一种附属）。

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>

    </class>

</hibernate-mapping>
```

注意，虽然是用目标类名来声明关联的，但是关联的目标类型除了是POJO之外，也可以 是一个动态的实体。

在使用 `dynamic-map` 为 `SessionFactory` 设置了默认的实体模式之后，可以在运行期使用 `Map` 的 `Map` 。

```
Session s = openSession();
Transaction tx = s.beginTransaction();
Session s = openSession();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();
```

动态映射的好处是，变化所需要的时间少了，因为原型不需要实现实体类。然而，你无法进行 编译期的类型检查，并可能由此会处理很多的运行期异常。幸亏有了Hibernate映射，它使得数 据库的schema能容易的规格化和合理化，并允许稍后在此之上添加合适的领域模型实现。

实体表示模式也能在每个 `Session` 的基础上设置：

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

请注意，用 `EntityMode` 调用 `getSession()` 是在 `Session` 的API中，而不是 `SessionFactory` 。这样，新的 `Session` 共享底层的JDBC连接，事务，和其他的上下文信 息。这意味着，你不需要在第二个 `Session` 中调用 `flush()` 和 `close()` ，同样的，把事务和连接的处理 交给原来的工作单元。

关于XML表示能力的更多信息可以在第 18 章 *XML*映射中找到。

4.5. 元组片断映射(Tuplizers)

`org.hibernate.tuple.Tuplizer` ，以及其子接口，负责根据给定的 `org.hibernate.EntityMode` ，来复现片断数据。如果给定的片断数据被认为其是一种数据结构，"tuplizer"就是一个知道如何创建这样的数据结构，以及如何给这个数据结构赋值的东西。比如说，对于POJO这种Entity Mode，对应的tuplizer知道通过其构造方法来创建一个POJO，再通过其属性访问器来访问POJO属性。有两大类高层Tuplizer，分别是 `org.hibernate.tuple.entity.EntityTuplizer` 和 `org.hibernate.tuple.entity.ComponentTuplizer` 接口。 `EntityTuplizer` 负责管理上面提到的实体的契约，而 `ComponentTuplizer` 则是针对组件的。

用户也可以插入其自定义的tuplizer。或许您需要一种不同于dynamic-map entity-mode中使用的 `java.util.HashMap` 的 `java.util.Map` 实现；或许您需要与默认策略不同的代理生成策略(proxy generation strategy)。通过自定义tuplizer实现，这两个目标您都可以达到。Tuplizer定义被附加到它们期望管理的entity或者component映射中。回到我们的customer entity例子：

```
<hibernate-mapping>
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>

public class CustomMapTuplizerImpl
        extends org.hibernate.tuple.entity.DynamicMapEntityTupli
zer {
    // override the buildInstantiator() method to plug in our cu
stom map...
    protected final Instantiator buildInstantiator(
            org.hibernate.mapping.PersistentClass mappingInfo) {
        return new CustomMapInstantiator( mappingInfo );
    }

    private static final class CustomMapInstantiator
            extends org.hibernate.tuple.DynamicMapInstantitor {
        // override the generateMap() method to return our custo
m map...
        protected final Map generateMap() {
            return new CustomMap();
        }
    }
}
```

第 5 章 对象/关系数据库映射基础(Basic O/R Mapping)

目录

5.1. 映射定义（Mapping declaration）

对象和关系数据库之间的映射通常是用一个XML文档(XML document)来定义的。这个映射文档被设计为易读的， 并且可以手工修改。映射语言是以Java为中心，这意味着映射文档是按照持久化类的定义来创建的， 而非表的定义。

请注意，虽然很多Hibernate用户选择手写XML映射文档，但也有一些工具可以用来生成映射文档， 包括XDoclet,Middlegen和AndroMDA。

让我们从一个映射的例子开始：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.
0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">

                <id name="id">
                        <generator class="native"/>
                </id>

                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>
```

```
                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>

                <subclass name="DomesticCat"
                    discriminator-value="D">

                        <property name="name"
                            type="string"/>

                </subclass>

        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

我们现在开始讨论映射文档的内容。我们只描述Hibernate在运行时用到的文档元素和属性。 映射文档还包括一些额外的可选属性和元素，它们在使用schema导出工具的时候会影响导出的数据库schema结果。（比如， `not-null` 属性。）

5.1.1. Doctype

所有的XML映射都需要定义如上所示的doctype。DTD可以从上述URL中获取， 也可以从 `hibernate-x.x.x/src/net/sf/hibernate` 目录中、或 `hibernate.jar` 文件中找到。Hibernate总是会首先在它的classptah中搜索DTD文件。 如果你发现它是通过连接Internet查找DTD文件，就对照你的classpath目录检查XML文件里的DTD声明。

5.1.1.1. EntityResolver

As mentioned previously, Hibernate will first attempt to resolve DTDs in its classpath. The manner in which it does this is by registering a custom `org.xml.sax.EntityResolver` implementation with the SAXReader it uses to read in the xml files. This custom `EntityResolver` recognizes two different systemId namespaces. 如前所述,Hibernate首先在其classpath中查找DTD。其行为是依靠在系统中注册的 `org.xml.sax.EntityResolver` 的一个具体实现，SAXReader依靠它来读取xml文件。这一 `EntityResolver` 实现能辨认两种不同的 systenId命名空间。

- 若resolver遇到了一个以 `http://hibernate.sourceforge.net/` 为开头的systemId，它会辨认出是 `hibernate namespace` ，resolver就试图通过加载Hibernate类的classloader来查找这些实体。
- 若resolver遇到了一个使用 `classpath://` URL协议的systemId，它会辨认出这是 `user namespace` ,resolver试图通过(1)当前线程上下文的classloader和(2)加载Hibernate class的classloader来查找这些实体。

使用user namespace(用户命名空间)的例子：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.
dtd" [
    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">
]>

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    <class>
    &types;
</hibernate-mapping>
```

`types.xml` 是 `your.domain` 包中的一个资源，它包含了一个定制的第 5.2.3 节 “自定义值类型”。

5.1.2. hibernate-mapping

这个元素包括一些可选的属性。 `schema` 和 `catalog` 属性， 指明了这个映射所连接（refer）的表所在的schema和/或catalog名称。 假若指定了这个属性，表名会加上所指定的schema和catalog的名字扩展为全限定名。假若没有指定，表名就不会使用全限定名。 `default-cascade` 指定了未明确注明 `cascade` 属性的Java属性和 集合类Hibernate会采取什么样的默认级联风格。 `auto-import` 属性默认让我们在查询语言中可以使用 非全限定名的类名。

```
<hibernate-mapping
         schema="schemaName"
         catalog="catalogName"
         default-cascade="cascade_style"
         default-access="field|property|ClassName"
         default-lazy="true|false"
         auto-import="true|false"
         package="package.name"
 />
```

❶	**`schema` (可选):** 数据库**schema**的名称。
❷	`catalog` (可选): 数据库catalog的名称。
❸	`default-cascade` (可选 - 默认为 `none`): 默认的级联风格。
❹	`default-access` (可选 - 默认为 `property`): Hibernate用来访问所有属性的策略。可以通过实现 `PropertyAccessor` 接口 自定义。
❺	`default-lazy` (可选 - 默认为 `true`): 指定了未明确注明 `lazy` 属性的Java属性和集合类， Hibernate会采取什么样的默认加载风格。
❻	`auto-import` (可选 - 默认为 `true`): 指定我们是否可以在查询语言中使用非全限定的类名（仅限于本映射文件中的类）。
❼	`package` (可选): 指定一个包前缀，如果在映射文档中没有指定全限定的类名， 就使用这个作为包名。

假若你有两个持久化类，它们的非全限定名是一样的（就是两个类的名字一样，所在的包不一样--译者注）， 你应该设置 `auto-import="false"` 。如果你把一个“import过”的名字同时对应两个类， Hibernate会抛出一个异常。

注意 `hibernate-mapping` 元素允许你嵌套多个如上所示的 <class> 映射。但是最好的做法（也许一些工具需要的）是一个 持久化类（或一个类的继承层次）对应一个映射文件，并以持久化的超类名称命名，例如： `Cat.hbm.xml` ， `Dog.hbm.xml` ，或者如果使用继承， `Animal.hbm.xml` 。

5.1.3. class

你可以使用 `class` 元素来定义一个持久化类：

```
<class
        name="ClassName"
        table="tableName"
        discriminator-value="discriminator_value"
        mutable="true|false"
        schema="owner"
        catalog="catalog"
        proxy="ProxyInterface"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        select-before-update="true|false"
        polymorphism="implicit|explicit"
        where="arbitrary sql where condition"
        persister="PersisterClass"
        batch-size="N"
        optimistic-lock="none|version|dirty|all"
        lazy="true|false"
        entity-name="EntityName"
        check="arbitrary sql check condition"
        rowid="rowid"
        subselect="SQL expression"
        abstract="true|false"
        node="element-name"
/>
```

❶	`name` (可选): 持久化类（或者接口）的Java全限定名。 如果这个属性不存在，Hibernate将假定这是一个非POJO的实体映射。
❷	`table` (可选 - 默认是类的非全限定名): 对应的数据库表名。
❸	`discriminator-value` (可选 - 默认和类名一样): 一个用于区分不同的子类的值，在多态行为时使用。它可以接受的值包括 `null` 和 `not null` 。
❹	`mutable` (可选，默认值为 `true`): 表明该类的实例是可变的或者不可变的。
❺	`schema` (可选): 覆盖在根 <hibernate-mapping> 元素中指定的schema名字。
❻	`catalog` (可选): 覆盖在根 <hibernate-mapping> 元素中指定的catalog名字。
❼	`proxy` (可选): 指定一个接口，在延迟装载时作为代理使用。 你可以在

❼	这里使用该类自己的名字。
❽	`dynamic-update` (可选, 默认为 `false`): 指定用于 `UPDATE` 的SQL 将会在运行时动态生成，并且只更新那些改变过的字段。
❾	`dynamic-insert` (可选, 默认为 `false`): 指定用于 `INSERT` 的 SQL 将会在运行时动态生成，并且只包含那些非空值字段。
❿	`select-before-update` (可选, 默认为 `false`): 指定Hibernate除非确定对象真正被修改了（如果该值为true－译注），否则不会执行SQL `UPDATE` 操作。在特定场合（实际上，它只在一个瞬时对象（transient object）关联到一个 新的session中时执行的update()中生效），这说明Hibernate会在 `UPDATE` 之前执行一次额外的SQL `SELECT` 操作，来决定是否应该执行 `UPDATE` 。
⓫	`polymorphism`（多态） (可选, 默认值为 `implicit (隐式)`): 界定是隐式还是显式的使用多态查询（这只在Hibernate的具体表继承策略中用到－译注）。
⓬	`where` (可选) 指定一个附加的SQL `WHERE` 条件， 在抓取这个类的对象时会一直增加这个条件。
⓭	`persister` (可选): 指定一个定制的 `ClassPersister` 。
⓮	`batch-size` (可选,默认是 `1`) 指定一个用于 根据标识符（identifier）抓取实例时使用的"batch size"（批次抓取数量）。
⓯	`optimistic-lock`（乐观锁定） (可选，默认是 `version`): 决定乐观锁定的策略。
(16)	`lazy` (可选): 通过设置 `lazy="false"` ， 所有的延迟加载（Lazy fetching）功能将被全部禁用（disabled）。
(17)	`entity-name` (可选，默认为类名): Hibernate3允许一个类进行多次映射（ 前提是映射到不同的表），并且允许使用Maps或XML代替Java层次的实体映射 （也就是实现动态领域模型，不用写持久化类－译注）。更多信息请看第 4.4 节 “动态模型(Dynamic models)” and 第 18 章 *XML 映射*。
(18)	`check` (可选): 这是一个SQL表达式， 用于为自动生成的schema添加多行（multi-row）约束检查。
(19)	`rowid` (可选): Hibernate可以使用数据库支持的所谓的ROWIDs，例如： Oracle数据库，如果你设置这个可选的 `rowid` ， Hibernate可以使用额外的字段 `rowid` 实现快速更新。ROWID是这个功能实现的重点， 它代表了一个存储元组（tuple）的物理位置。
(20)	`subselect` (可选): 它将一个不可变（immutable）并且只读的实体映射到一个数据库的 子查询中。当你想用视图代替一张基本表的时候，这是有用的，但最好不要这样做。更多的介绍请看下面内容。
	`abstract` (可选): 用于在 `<union-subclass>` 的继承结构

	（hierarchies）中标识抽象超类。

若指明的持久化类实际上是一个接口，这也是完全可以接受的。 之后你可以用元素 <subclass> 来指定该接口的实际实现类。 你可以持久化任何*static*（静态的）内部类。 你应该使用标准的类名格式来指定类名， 比如：Foo$Bar 。

不可变类， mutable="false" 不可以被应用程序更新或者删除。 这可以让Hibernate做一些小小的性能优化。

可选的 proxy 属性允许延迟加载类的持久化实例。 Hibernate开始会返回实现了这个命名接口的CGLIB代理。当代理的某个方法被实际调用的时候， 真实的持久化对象才会被装载。参见下面的“用于延迟装载的代理”。

Implicit (隐式)的多态是指，如果查询时给出的是任何超类、该类实现的接口或者该类的 名字，都会返回这个类的实例；如果查询中给出的是子类的名字，则会返回子类的实例。 *Explicit* （显式）的多态是指，只有在查询时给出明确的该类名字时才会返回这个类的实例； 同时只有在这个 <class> 的定义中作为 <subclass> 或者 <joined-subclass> 出现的子类，才会可能返回。 在大多数情况下，默认的 polymorphism="implicit" 都是合适的。 显式的多态在有两个不同的类映射到同一个表的时候很有用。（允许一个“轻型”的类，只包含部分表字段）。

persister 属性可以让你定制这个类使用的持久化策略。 你可以指定你自己实现 org.hibernate.persister.EntityPersister 的子类，你甚至可以完全从头开始编写一个 org.hibernate.persister.ClassPersister 接口的实现， 比如是用储存过程调用、序列化到文件或者LDAP数据库来实现。 参阅 org.hibernate.test.CustomPersister ，这是一个简单的例子 （“持久化”到一个 Hashtable ）。

请注意 dynamic-update 和 dynamic-insert 的设置并不会继承到子类， 所以在 <subclass> 或者 <joined-subclass> 元素中可能 需要再次设置。这些设置是否能够提高效率要视情形而定。请用你的智慧决定是否使用。

使用 select-before-update 通常会降低性能。如果你重新连接一个脱管（detache）对象实例 到一个 Session 中时，它可以防止数据库不必要的触发update。 这就很有用了。

如果你打开了 dynamic-update ，你可以选择几种乐观锁定的策略：

- version（版本检查） 检查version/timestamp字段
- all（全部） 检查全部字段
- dirty（脏检查） 只检察修改过的字段
- none（不检查） 不使用乐观锁定

我们非常强烈建议你在Hibernate中使用version/timestamp字段来进行乐观锁定。 对性能来说，这是最好的选择，并且这也是唯一能够处理在session外进行操作的策略（例如： 在使用 Session.merge() 的时候）。

对Hibernate映射来说视图和表是没有区别的，这是因为它们在数据层都是透明的（注意：一些数据库不支持视图属性，特别是更新的时候）。有时你想使用视图，但却不能在数据库 中创建它（例如：在遗留的schema中）。这样的话，你可以映射一个不可变的（immutable）并且是 只读的实体到一个给定的SQL子查询表达式：

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

定义这个实体用到的表为同步（synchronize），确保自动刷新（auto-flush）正确执行， 并且依赖原实体的查询不会返回过期数据。`<subselect>` 在属性元素 和一个嵌套映射元素中都可见。

5.1.4. id

被映射的类必须定义对应数据库表主键字段。大多数类有一个JavaBeans风格的属性， 为每一个实例包含唯一的标识。 `<id>` 元素定义了该属性到数据库表主键字段的映射。

```
<id
        name="propertyName"
        type="typename"
        column="column_name"
        unsaved-value="null|any|none|undefined|id_value"
        access="field|property|ClassName"
        node="element-name|@attribute-name|element/@attribute|."
>

        <generator class="generatorClass"/>
</id>
```

❶	`name` **(可选):** 标识属性的名字。
❷	`type` (可选): 标识Hibernate类型的名字。
❸	`column` (可选 - 默认为属性名): 主键字段的名字。
❹	`unsaved-value` (可选 - 默认为一个切合实际（sensible）的值): 一个特定的标识属性值，用来标志该实例是刚刚创建的，尚未保存。 这可以把这种实例和从以前的session中装载过（可能又做过修改--译者注） 但未再次持久化的实例区分开来。
❺	`access` (可选 - 默认为 `property`): Hibernate用来访问属性值的策略。

如果 `name` 属性不存在，会认为这个类没有标识属性。

`unsaved-value` 属性在Hibernate3中几乎不再需要。

还有一个另外的 `<composite-id>` 定义可以访问旧式的多主键数据。 我们强烈不建议使用这种方式。

5.1.4.1. Generator

可选的 `<generator>` 子元素是一个Java类的名字， 用来为该持久化类的实例生成唯一的标识。如果这个生成器实例需要某些配置值或者初始化参数， 用 `<param>` 元素来传递。

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param
>
        </generator>
</id>
```

所有的生成器都实现 `org.hibernate.id.IdentifierGenerator` 接口。 这是一个非常简单的接口；某些应用程序可以选择提供他们自己特定的实现。当然，Hibernate提供了很多内置的实现。下面是一些内置生成器的快捷名字：

`increment`

用于为 `long` , `short` 或者 `int` 类型生成 唯一标识。只有在没有其他进程往同一张表中插入数据时才能使用。 在集群下不要使用。

`identity`

对DB2,MySQL, MS SQL Server, Sybase和HypersonicSQL的内置标识字段提供支持。 返回的标识符是 `long` , `short` 或者 `int` 类型的。

`sequence`

在DB2,PostgreSQL, Oracle, SAP DB, McKoi中使用序列（sequence)， 而在Interbase中使用生成器(generator)。返回的标识符是 `long` , `short` 或者 `int` 类型的。

`hilo`

使用一个高/低位算法高效的生成 `long` , `short` 或者 `int` 类型的标识符。给定一个表和字段（默认分别是 `hibernate_unique_key` 和 `next_hi` ）作为高位值的来源。 高/低位算法生成的标识符只在一个特定的数据库中是唯一的。

`seqhilo`

使用一个高/低位算法来高效的生成 `long` , `short` 或者 `int` 类型的标识符，给定一个数据库序列（sequence)的名字。

`uuid`

用一个128-bit的UUID算法生成字符串类型的标识符， 这在一个网络中是唯一的（使用了IP地址）。UUID被编码为一个32位16进制数字的字符串。

`guid`

在MS SQL Server 和 MySQL 中使用数据库生成的GUID字符串。

`native`

根据底层数据库的能力选择 `identity`, `sequence` 或者 `hilo` 中的一个。

`assigned`

让应用程序在 `save()` 之前为对象分配一个标示符。这是 `<generator>` 元素没有指定时的默认生成策略。

`select`

通过数据库触发器选择一些唯一主键的行并返回主键值来分配一个主键。

`foreign`

使用另外一个相关联的对象的标识符。通常和 `<one-to-one>` 联合起来使用。

`sequence-identity`

一种特别的序列生成策略,使用数据库序列来生成实际值,但将它和JDBC3的getGeneratedKeys结合在一起,使得在插入语句执行的时候就返回生成的值。目前为止只有面向JDK 1.4的Oracle 10g驱动支持这一策略。注意，因为Oracle驱动程序的一个bug，这些插入语句的注释被关闭了。（原文：Note comments on these insert statements are disabled due to a bug in the Oracle drivers.）

5.1.4.2. 高/低位算法（Hi/Lo Algorithm）

`hilo` 和 `seqhilo` 生成器给出了两种hi/lo算法的实现， 这是一种很令人满意的标识符生成算法。第一种实现需要一个“特殊”的数据库表来保存下一个可用的“hi”值。 第二种实现使用一个Oracle风格的序列（在被支持的情况下）。

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

很不幸，你在为Hibernate自行提供 `Connection` 时无法使用 `hilo` 。 当Hibernate使用JTA获取应用服务器的数据源连接时,你必须正确地配置 `hibernate.transaction.manager_lookup_class` 。

5.1.4.3. UUID算法（UUID Algorithm ）

UUID包含：IP地址，JVM的启动时间（精确到1/4秒），系统时间和一个计数器值（在JVM中唯一）。 在Java代码中不可能获得MAC地址或者内存地址，所以这已经是我们在不使用JNI的前提下的能做的最好实现了。

5.1.4.4. 标识字段和序列（Identity columns and Sequences）

对于内部支持标识字段的数据库(DB2,MySQL,Sybase,MS SQL)，你可以使用 `identity` 关键字生成。 对于内部支持序列的数据库（DB2,Oracle, PostgreSQL, Interbase, McKoi,SAP DB), 你可以使用 `sequence` 风格的关键字生成。 这两种方式对于插入一个新的对象都需要两次SQL查询。

```
<id name="id" type="long" column="person_id">
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param
>
        </generator>
</id>
```

```
<id name="id" type="long" column="person_id" unsaved-value="0">
        <generator class="identity"/>
</id>
```

对于跨平台开发， `native` 策略会从 `identity` ， `sequence` 和 `hilo` 中进行选择，选择哪一个，这取决于底层数据库的支持能力。

5.1.4.5. 程序分配的标识符（Assigned Identifiers）

如果你需要应用程序分配一个标示符（而非Hibernate来生成），你可以使用 assigned 生成器。这种特殊的生成器会使用已经分配给对象的标识符属性的标识符值。 这个生成器使用一个自然键（natural key，有商业意义的列－译注）作为主键，而不是使用一个代理键（ surrogate key，没有商业意义的列－译注）。这是没有指定 <generator> 元素时的默认行为

当选择 assigned 生成器时，除非有一个version或timestamp属性，或者你定义了 Interceptor.isUnsaved() ，否则需要让Hiberante使用 unsaved-value="undefined" ，强制Hibernatet查询数据库来确定一个实例是瞬时的（transient） 还是脱管的（detached）。

5.1.4.6. 触发器实现的主键生成器（Primary keys assigned by triggers）

仅仅用于遗留的schema中 (Hibernate不能使用触发器生成DDL)。

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

在上面的例子中，类定义了一个命名为 `socialSecurityNumber` 的唯一值属性，它是一个自然键（natural key），命名为 `person_id` 的代理键（surrogate key）的值由触发器生成。

5.1.5. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName"
        node="element-name|."
        >

        <key-property name="propertyName" type="typename" column
="column_name"/>
        <key-many-to-one name="propertyName class="ClassName" co
lumn="column_name"/>
        ......
</composite-id>
```

如果表使用联合主键，你可以映射类的多个属性为标识符属性。 `<composite-id>` 元素接受 `<key-property>` 属性映射和 `<key-many-to-one>` 属性映射作为子元素。

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

你的持久化类必须重载 `equals()` 和 `hashCode()` 方法，来实现组合的标识符的相等判断。 实现 `Serializable` 接口也是必须的。

不幸的是，这种组合关键字的方法意味着一个持久化类是它自己的标识。除了对象自己之外， 没有什么方便的“把手”可用。你必须初始化持久化类的实例，填充它的标识符属性，再 `load()` 组合关键字关联的持久状态。我们把这种方法称为 *embedded*（嵌入式）的组合标识符，在重要的应用中不鼓励使用这种用法。

第二种方法我们称为*mapped(*映射式*)*组合标识符 (mapped composite identifier), `<composite-id>` 元素中列出的标识属性不但在持久化类出现，还形成一个独立的标识符类。

```
<composite-id class="MedicareId" mapped="true">
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

在这个例子中，组合标识符类 `MedicareId` 和实体类都含有 `medicareNumber` 和 `dependent` 属性。标识符类必须重载 `equals()` 和 `hashCode()` 并且实现 `Serializable` 接口。这种方法的缺点是出现了明显的代码重复。

下面列出的属性是用来指定一个映射式组合标识符的：

- `mapped` (可选, 默认为 `false`): 指明使用一个映射式组合标识符，其包含的属性映射同时在实体类和组合标识符类中出现。
- `class` (可选,但对映射式组合标识符必须指定): 作为组合标识符类使用的类名.

在第 8.4 节 “组件作为联合标识符(Components as composite identifiers)”一节中,我们会描述第三种方式,那就是把组合标识符实现为一个组件(component)类,这是更方便的方法。下面的属性仅对第三种方法有效：

- `name` (可选,但对这种方法而言必须): 包含此组件标识符的组件类型的名字 (参阅第9章).
- `access` (可选 - 默认为 `property`): Hibernate应该使用的访问此属性值的策略
- `class` (可选 - 默认会用反射来自动判定属性类型): 用来作为组合标识符的组件类的类名（参阅下一节）

第三种方式，被称为*identifier component(*标识符组件*)*是我们对几乎所有应用都推荐使用的方式。

5.1.6. 鉴别器（discriminator）

在"一棵对象继承树对应一个表"的策略中, `<discriminator>` 元素是必需的, 它定义了表的鉴别器字段。鉴别器字段包含标志值，用于告知持久化层应该为某个特定的行创建哪一个子类的实例。 如下这些受到限制的类型可以使用： `string` , `character` , `integer` , `byte` , `short` , `boolean` , `yes_no` , `true_false` .

```
<discriminator
        column="discriminator_column"
        type="discriminator_type"
        force="true|false"
        insert="true|false"
        formula="arbitrary sql expression"
/>
```

❶	**`column`** **(**可选 **-** 默认为 **`class`** **)** 鉴别器字段的名字
❷	`type` (可选 - 默认为 `string`) 一个Hibernate字段类型的名字
❸	`force(强制)` (可选 - 默认为 `false`) "强制"Hibernate指定允许的鉴别器值,即使当取得的所有实例都是根类的。
❹	`insert` (可选 - 默认为 `true`) 如果你的鉴别器字段也是映射为复合标识（composite identifier）的一部分，则需将 这个值设为 `false` 。（告诉Hibernate在做SQL `INSERT` 时不包含该列）
❺	`formula` (可选) 一个SQL表达式，在类型判断（判断是父类还是具体子类一译注）时执行。可用于基于内容的鉴别器。

鉴别器字段的实际值是根据 `<class>` 和 `<subclass>` 元素中的 `discriminator-value` 属性得来的。

`force` 属性仅仅在这种情况下有用的：表中包含没有被映射到持久化类的附加辨别器值。 这种情况不会经常遇到。

使用 `formula` 属性你可以定义一个SQL表达式，用来判断一个行数据的类型。

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else
 1 end"
    type="integer"/>
```

5.1.7. 版本（version）(可选)

`<version>` 元素是可选的，表明表中包含附带版本信息的数据。 这在你准备使用 长事务（*long transactions*）的时候特别有用。（见后）

```
<version
        column="version_column"
        name="propertyName"
        type="typename"
        access="field|property|ClassName"
        unsaved-value="null|negative|undefined"
        generated="never|always"
        insert="true|false"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

❶	**`column` (可选 - 默认为属性名): 指定持有版本号的字段名。**
❷	`name` : 持久化类的属性名。
❸	`type` (可选 - 默认是 `integer`): 版本号的类型。
❹	`access` (可选 - 默认是 `property`): Hibernate用于访问属性值的策略。
❺	`unsaved-value` (可选 - 默认是 `undefined`): 用于标明某个实例时刚刚被实例化的（尚未保存）版本属性值，依靠这个值就可以把这种情况 和已经在先前的session中保存或装载的脱管（detached）实例区分开来。（ `undefined` 指明应被使用的标识属性值。）
❻	`generated` (可选 - 默认是 `never`): 表明此版本属性值是否实际上是由数据库生成的。请参阅第 5.6 节 “数据库生成属性（Generated Properties）”部分的讨论。
❼	`insert` (可选 - 默认是 `true`): 表明此版本列应该包含在SQL插入语句中。只有当数据库字段有默认值 `0` 的时候，才可以设置为 `false` 。

版本号必须是以下类型： `long` , `integer` , `short` , `timestamp` 或者 `calendar` 。

一个脱管（detached）实例的version或timestamp属性不能为空（null），因为Hibernate不管 `unsaved-value` 被指定为何种策略，它将任何属性为空的version或timestamp 实例看作为瞬时（transient）实例。 避免*Hibernate*中的传递重附（*transitive reattachment*）问题的一个简单方法是 定义一个不能为空的*version*或*timestamp*属性，特别是在人们使用程序分配的标识符（*assigned identifiers*） 或复合主键时非常有用！

5.1.8. timestamp (可选)

可选的 `<timestamp>` 元素指明了表中包含时间戳数据。 这用来作为版本的替代。时间戳本质上是一种对乐观锁定的一种不是特别安全的实现。当然， 有时候应用程序可能在其他方面使用时间戳。

```
<timestamp
        column="timestamp_column"
        name="propertyName"
        access="field|property|ClassName"
        unsaved-value="null|undefined"
        source="vm|db"
        generated="never|always"
        node="element-name|@attribute-name|element/@attribute|."
/>
```

❶	**`column` (可选 - 默认为属性名): 持有时间戳的字段名。**
❷	`name` : 在持久化类中的JavaBeans风格的属性名， 其Java类型是 `Date` 或者 `Timestamp` 的。
❸	`access` (可选 - 默认是 `property`): Hibernate用于访问属性值的策略。
❹	`unsaved-value` (可选 - 默认是 `null`): 用于标明某个实例时刚刚被实例化的（尚未保存）版本属性值，依靠这个值就可以把这种情况和 已经在先前的session中保存或装载的脱管（detached）实例区分开来。（ `undefined` 指明使用标识属性值进行这种判断。）
❺	`source` (可选 - 默认是 `vm`): Hibernate如何才能获取到时间戳的值呢？从数据库，还是当前JVM？从数据库获取会带来一些负担，因为Hibernate必须访问数据库来获得“下一个值”，但是在集群环境中会更安全些。还要注意，并不是所有的 `Dialect`（方言） 都支持获得数据库的当前时间戳的，而支持的数据库中又有一部分因为精度不足,用于锁定是不安全的（例如 Oracle 8）。
❻	`generated` (可选 - 默认是 `never`): 指出时间戳值是否实际上是由数据库生成的.请参阅第 5.6 节 “数据库生成属性（Generated Properties）”的讨论。

注意， `<timestamp>` 和 `<version type="timestamp">` 是等价的。并且 `<timestamp source="db">` 和 `<version type="dbtimestamp"&gt` 是等价的。

5.1.9. property

`<property>` 元素为类定义了一个持久化的,JavaBean风格的属性。

```
<property
        name="propertyName"
        column="column_name"
        type="typename"
        update="true|false"
        insert="true|false"
        formula="arbitrary SQL expression"
        access="field|property|ClassName"
        lazy="true|false"
        unique="true|false"
        not-null="true|false"
        optimistic-lock="true|false"
        generated="never|insert|always"
        node="element-name|@attribute-name|element/@attribute|."

        index="index_name"
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
/>
```

❶	`name`：属性的名字,以小写字母开头。
❷	`column` (可选 - 默认为属性名字): 对应的数据库字段名。 也可以通过嵌套的 `<column>` 元素指定。
❸	`type` (可选): 一个Hibernate类型的名字。
❹	`update, insert` (可选 - 默认为 `true`)：表明用于 `UPDATE` 和/或 `INSERT` 的SQL语句中是否包含这个被映射了的字段。这二者如果都设置为 `false` 则表明这是一个“外源性（derived）”的属性，它的值来源于映射到同一个（或多个） 字段的某些其他属性，或者通过一个trigger(触发器）或其他程序生成。
❺	`formula` (可选): 一个SQL表达式，定义了这个计算 （*computed*） 属性的值。计算属性没有和它对应的数据库字段。
❻	`access` (可选 - 默认值为 `property`): Hibernate用来访问属性值的策略。
❼	`lazy` (可选 - 默认为 `false`): 指定 指定实例变量第一次被访问时，这个属性是否延迟抓取（fetched lazily）（ 需要运行时字节码增强）。
❽	`unique` (可选): 使用DDL为该字段添加唯一的约束。 同样，允许它作为 `property-ref` 引用的目标。
❾	`not-null` (可选): 使用DDL为该字段添加可否为空（nullability）的约束。
❿	`optimistic-lock` (可选 - 默认为 `true`): 指定这个属性在做更新时是否需要获得乐观锁定（optimistic lock）。 换句话说，它决定这个属性发生脏数据时版本（version）的值是否增长。
⓫	`generated` (可选 - 默认为 `never`): 表明此属性值是否实际上是由数据库生成的。请参阅第 5.6 节 “数据库生成属性（Generated Properties）”的讨论。

*typename*可以是如下几种：

1. Hibernate基本类型名（比如： `integer, string, character,date, timestamp, float, binary, s` ）。
2. 一个Java类的名字，这个类属于一种默认基础类型 (比如： `int, float,char, java.lang.String, java.util.Date, java.lang.Int`)。
3. 一个可以序列化的Java类的名字。
4. 一个自定义类型的类的名字。（比如： `com.illflow.type.MyCustomType`)。

如果你没有指定类型，Hibernarte会使用反射来得到这个名字的属性，以此来猜测正确的Hibernate类型。 Hibernate会按照规则2,3,4的顺序对属性读取器(getter方法）的返回类进行解释。然而，这还不够。 在某些情况下你仍然需要 `type` 属性。（比如，为了区别 `Hibernate.DATE` 和 `Hibernate.TIMESTAMP` ,或者为了指定一个自定义类型。）

`access` 属性用来让你控制Hibernate如何在运行时访问属性。在默认情况下，Hibernate会使用属性的get/set方法对（pair）。如果你指明 `access="field"` ,Hibernate会忽略get/set方法对，直接使用反射来访问成员变量。你也可以指定你自己的策略， 这就需要你自己实现 `org.hibernate.property.PropertyAccessor` 接口， 再在access中设置你自定义策略类的名字。

衍生属性（derive propertie）是一个特别强大的特征。这些属性应该定义为只读，属性值在装载时计算生成。 你用一个SQL表达式生成计算的结果，它会在这个实例转载时翻译成一个SQL查询的 `SELECT` 子查询语句。

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li
, Product p
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

注意，你可以使用实体自己的表，而不用为这个特别的列定义别名（ 上面例子中的 `customerId` ）。同时注意，如果你不喜欢使用属性， 你可以使用嵌套的 `<formula>` 映射元素。

5.1.10. 多对一（many-to-one）

通过 `many-to-one` 元素,可以定义一种常见的与另一个持久化类的关联。 这种关系模型是多对一关联（实际上是一个对象引用－译注）：这个表的一个外键引用目标表的 主键字段。

```
<many-to-one
        name="propertyName"
        column="column_name"
        class="ClassName"
        cascade="cascade_style"
        fetch="join|select"
        update="true|false"
        insert="true|false"
        property-ref="propertyNameFromAssociatedClass"
        access="field|property|ClassName"
        unique="true|false"
        not-null="true|false"
        optimistic-lock="true|false"
        lazy="proxy|no-proxy|false"
        not-found="ignore|exception"
        entity-name="EntityName"
        formula="arbitrary SQL expression"
        node="element-name|@attribute-name|element/@attribute|."

        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"

/>
```

❶	**name** : 属性名。
❷	`column` (可选): 外间字段名。它也可以通过嵌套的 `<column>` 元素指定。
❸	`class` (可选 - 默认是通过反射得到属性类型): 关联的类的名字。
❹	`cascade（级联）` (可选): 指明哪些操作会从父对象级联到关联的对象。
❺	`fetch` (可选 - 默认为 `select`): 在外连接抓取（outer-join fetching）和序列选择抓取（sequential select fetching）两者中选择其一。
❻	`update, insert` (可选 - 默认为 `true`) 指定对应的字段是否包含在用于 `UPDATE` 和/或 `INSERT` 的SQL语句中。如果二者都是 `false` ,则这是一个纯粹的“外源性（derived）”关联，它的值是通过映射到同一个（或多个）字段的某些其他属性得到 或者通过trigger(触发器）、或其他程序生成。
❻	`property-ref` :(可选) 指定关联类的一个属性，这个属性将会和本外键相对应。 如果没有指定，会使用对方关联类的主键。
❼	`access` (可选 - 默认是 `property`): Hibernate用来访问属性的策略。
❽	`unique` (可选): 使用DDL为外键字段生成一个唯一约束。此外， 这也可以用作 `property-ref` 的目标属性。这使关联同时具有 一对一的效果。
❾	`not-null` (可选): 使用DDL为外键字段生成一个非空约束。
❿	`optimistic-lock` (可选 - 默认为 `true`): 指定这个属性在做更新时是否需要获得乐观锁定（optimistic lock）。 换句话说，它决定这个属性发生脏数据时版本（version）的值是否增长。
⓫	`lazy` (可选 - 默认为 `proxy`): 默认情况下，单点关联是经过代理的。 `lazy="no-proxy"` 指定此属性应该在实例变量第一次被访问时应该延迟抓取（fetche lazily）（需要运行时字节码的增强）。 `lazy="false"` 指定此关联总是被预先抓取。
⓬	`not-found` (可选 - 默认为 `exception`): 指定外键引用的数据不存在时如何处理： `ignore` 会将行数据不存在视为一个空（null）关联。
⓭	`entity-name` (可选): 被关联的类的实体名。
⓮	`formula` (可选): SQL表达式，用于定义*computed*（计算出的）外键值。

`cascade` 属性设置为除了 `none` 以外任何有意义的值， 它将把特定的操作传递到关联对象中。这个值就代表着Hibernate基本操作的名称， `persist, merge, delete, save-update, evict, replicate, lock, refresh` ，以及特别的值 `delete-orphan` 和 `all` ，并且可以用逗号分隔符 来组合这些操作，例如， `cascade="persist,merge,evict"` 或 `cascade="all,delete-orphan"` 。更全面的解释请参考第 10.11 节 “传播性持久化(transitive persistence)”. 注意，单值关联 (many-to-one 和 one-to-one 关联) 不支持删除孤儿（orphan delete，删除不再被引用的值）.

一个典型的简单 `many-to-one` 定义例子：

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/
>
```

`property-ref` 属性只应该用来对付遗留下来的数据库系统， 可能有外键指向对方关联表的是个非主键字段（但是应该是一个惟一关键字）的情况下。 这是一种十分丑陋的关系模型。比如说，假设 `Product` 类有一个惟一的序列号， 它并不是主键。（ `unique` 属性控制Hibernate通过SchemaExport工具进行的DDL生成。）

```
<property name="serialNumber" unique="true" type="string" column
="SERIAL_NUMBER"/>
```

那么关于 `OrderItem` 的映射可能是：

```
<many-to-one name="product" property-ref="serialNumber" column="
PRODUCT_SERIAL_NUMBER"/>
```

当然，我们决不鼓励这种用法。

如果被引用的唯一主键由关联实体的多个属性组成，你应该在名称为 <properties> 的元素 里面映射所有关联的属性。

假若被引用的唯一主键是组件的属性，你可以指定属性路径：

```
<many-to-one name="owner" property-ref="identity.ssn" column="OW
NER_SSN"/>
```

5.1.11. 一对一

持久化对象之间一对一的关联关系是通过 `one-to-one` 元素定义的。

```
<one-to-one
        name="propertyName"
        class="ClassName"
        cascade="cascade_style"
        constrained="true|false"
        fetch="join|select"
        property-ref="propertyNameFromAssociatedClass"
        access="field|property|ClassName"
        formula="any SQL expression"
        lazy="proxy|no-proxy|false"
        entity-name="EntityName"
        node="element-name|@attribute-name|element/@attribute|."

        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

❶	`name`：属性的名字。
❷	`class` (可选 - 默认是通过反射得到的属性类型)：被关联的类的名字。
❸	`cascade(级联)` (可选) 表明操作是否从父对象级联到被关联的对象。
❹	`constrained(约束)` (可选) 表明该类对应的表对应的数据库表，和被关联的对象所对应的数据库表之间，通过一个外键引用对主键进行约束。 这个选项影响 `save()` 和 `delete()` 在级联执行时的先后顺序以及 决定该关联能否被委托(也在schema export tool中被使用).
❺	`fetch` (可选 - 默认设置为 选择): 在外连接抓取或者序列选择抓取选择其一.
❻	`property-ref`: (可选) 指定关联类的属性名，这个属性将会和本类的主键相对应。如果没有指定，会使用对方关联类的主键。
❼	`access` (可选 - 默认是 `property`): Hibernate用来访问属性的策略。
❽	`formula` (可选):绝大多数一对一的关联都指向其实体的主键。在一些少见的情况中， 你可能会指向其他的一个或多个字段，或者是一个表达式，这些情况下，你可以用一个SQL公式来表示。 （可以在org.hibernate.test.onetooneformula找到例子）
❾	`lazy` (可选 - 默认为 `proxy`): 默认情况下，单点关联是经过代理的。 `lazy="no-proxy"` 指定此属性应该在实例变量第一次被访问时应该延迟抓取（fetche lazily）（需要运行时字节码的增强）。 `lazy="false"` 指定此关联总是被预先抓取。注意，如果 *constrained="false"* , 不可能使用代理，*Hibernate*会采取预先抓取！
❿	`entity-name` (可选): 被关联的类的实体名。

有两种不同的一对一关联：

- 主键关联
- 惟一外键关联

主键关联不需要额外的表字段；如果两行是通过这种一对一关系相关联的，那么这两行就共享同样的主关键字值。所以如果你希望两个对象通过主键一对一关联，你必须确认它们被赋予同样的标识值！

比如说，对下面的 `Employee` 和 `Person` 进行主键一对一关联:

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/
>
```

现在我们必须确保PERSON和EMPLOYEE中相关的字段是相等的。我们使用一个被成为 `foreign` 的特殊的hibernate标识符生成策略：

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

一个刚刚保存的 `Person` 实例被赋予和该 `Person` 的 `employee` 属性所指向的 `Employee` 实例同样的关键字值。

另一种方式是一个外键和一个惟一关键字对应，上面的 `Employee` 和 `Person` 的例子，如果使用这种关联方式，可以表达成：

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique="true"/>
```

如果在 `Person` 的映射加入下面几句，这种关联就是双向的：

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

5.1.12. 自然ID(natural-id)

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

我们建议使用代用键（键值不具备实际意义）作为主键，我们仍然应该尝试为所有的实体采用自然的键值作为（附加——译者注）标示。自然键（natural key）是单个或组合属性，他们必须唯一且非空。如果它还是不可变的那就更理想了。在 `<natural-id>` 元素中列出自然键的属性。Hibernate会帮你生成必须的唯一键值和非空约束，你的映射会更加的明显易懂（原文是self-documenting，自我注解）。

我们强烈建议你实现 `equals()` 和 `hashCode()` 方法,来比较实体的自然键属性。

这一映射不是为了把自然键作为主键而准备的。

- `mutable` (可选, 默认为 `false`): 默认情况下，自然标识属性被假定为不可变的（常量）。

5.1.13. 组件(component), 动态组件(dynamic-component)

<component> 元素把子对象的一些元素与父类对应的表的一些字段映射起来。 然后组件可以定义它们自己的属性、组件或者集合。参见后面的“Components”一章。

```
<component
        name="propertyName"
        class="className"
        insert="true|false"
        update="true|false"
        access="field|property|ClassName"
        lazy="true|false"
        optimistic-lock="true|false"
        unique="true|false"
        node="element-name|."
>

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

❶	**name** : 属性名
❷	class (可选 - 默认为通过反射得到的属性类型):组件(子)类的名字。
❸	insert : 被映射的字段是否出现在SQL的 INSERT 语句中?
❹	update : 被映射的字段是否出现在SQL的 UPDATE 语句中?
❺	access (可选 - 默认是 property): Hibernate用来访问属性的策略。
❻	lazy (可选 - 默认是 false): 表明此组件应在实例变量第一次被访问的时候延迟加载(需要编译时字节码装置器)
❼	optimistic-lock (可选 - 默认是 true):表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号(Version)
❽	unique (可选 - 默认是 false):表明组件映射的所有字段上都有唯一性约束

其 <property> 子标签为子类的一些属性与表字段之间建立映射。

<component> 元素允许加入一个 <parent> 子元素，在组件类内部就可以有一个指向其容器的实体的反向引用。

`<dynamic-component>`元素允许把一个`Map`映射为组件，其属性名对应map的键值。参见第 8.5 节“动态组件（Dynamic components）”.

5.1.14. properties

<properties> 元素允许定义一个命名的逻辑分组(grouping)包含一个类中的多个属性。 这个元素最重要的用处是允许多个属性的组合作为 `property-ref` 的目标(target)。 这也是定义多字段唯一约束的一种方便途径。

```
<properties
        name="logicalName"
        insert="true|false"
        update="true|false"
        optimistic-lock="true|false"
        unique="true|false"
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

❶	**name**: 分组的逻辑名称 - 不是 实际属性的名称.
❷	`insert`: 被映射的字段是否出现在SQL的 `INSERT` 语句中?
❸	`update`: 被映射的字段是否出现在SQL的 `UPDATE` 语句中?
❹	`optimistic-lock` (可选 - 默认是 `true`):表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号(Version)
❺	`unique` (可选 - 默认是 `false`):表明组件映射的所有字段上都有唯一性约束

例如，如果我们有如下的 <properties> 映射:

```
<class name="Person">
    <id name="personNumber"/>
    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

然后，我们可能有一些遗留的数据关联，引用 `Person` 表的这个唯一键，而不是主键。

```
<many-to-one name="person"
         class="Person" property-ref="name">
    <column name="firstName"/>
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

我们并不推荐这样使用，除非在映射遗留数据的情况下。

5.1.15. 子类(subclass)

最后，多态持久化需要为父类的每个子类都进行定义。对于“每一棵类继承树对应一个表”的策略来说，就需要使用 <subclass> 定义。

```
<subclass
        name="ClassName"
        discriminator-value="discriminator_value"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

        <property .... />
        .....
</subclass>
```

•	**name** : 子类的全限定名。
•	discriminator-value(辨别标志) (可选 - 默认为类名):一个用于区分每个独立的子类的值。
•	proxy(代理) (可选): 指定一个类或者接口，在延迟装载时作为代理使用。
•	lazy (可选, 默认是 true): 设置为 lazy="false" 禁止使用延迟抓取

每个子类都应该定义它自己的持久化属性和子类。 <version> 和 <id> 属性可以从根父类继承下来。在一棵继承树上的每个子类都必须定义一个唯一的 discriminator-value 。如果没有指定，就会使用Java类的全限定名。

更多关于继承映射的信息, 参考 第 9 章 继承映射(*Inheritance Mappings*)章节.

5.1.16. 连接的子类(joined-subclass)

此外，每个子类可能被映射到他自己的表中(每个子类一个表的策略)。被继承的状态通过和超类的表关联得到。我们使用 `<joined-subclass>` 元素。

```
<joined-subclass
        name="ClassName"
        table="tablename"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

❶	**`name`**：子类的全限定名。
❷	`table`：子类的表名.
❸	`proxy` (可选): 指定一个类或者接口，在延迟装载时作为代理使用。
❹	`lazy` (可选, 默认是 `true`): 设置为 `lazy="false"` 禁止使用延迟装载。

这种映射策略不需要指定辨别标志(discriminator)字段。但是，每一个子类都必须使用 `<key>` 元素指定一个表字段来持有对象的标识符。本章开始的映射可以被用如下方式重写：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.
dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMES
TIC_CATS">
                    <key column="CAT"/>
                    <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

更多关于继承映射的信息，参考第 9 章 继承映射(*Inheritance Mappings*)。

5.1.17. 联合子类(union-subclass)

第三种选择是仅仅映射类继承树中具体类部分到表中(每个具体类一张表的策略)。其中，每张表定义了类的所有持久化状态，包括继承的状态。在 Hibernate 中，并不需要完全显式地映射这样的继承树。你可以简单地使用单独的 `<class>` 定义映射每个类。然而，如果你想使用多态关联(例如，一个对类继承树中超类的关联)，你需要使用 `<union-subclass>` 映射。

```
<union-subclass
        name="ClassName"
        table="tablename"
        proxy="ProxyInterface"
        lazy="true|false"
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <property .... />
        .....
</union-subclass>
```

❶	**`name`**：子类的全限定名。
❷	`table`：子类的表名
❸	`proxy` (可选): 指定一个类或者接口，在延迟装载时作为代理使用。
❹	`lazy` (可选, 默认是 `true`): 设置为 `lazy="false"` 禁止使用延迟装载。

这种映射策略不需要指定辨别标志(discriminator)字段。

更多关于继承映射的信息，参考第 9 章 继承映射(*Inheritance Mappings*)。

5.1.18. 连接(join)

使用 `<join>` 元素，假若在表之间存在一对一关联,可以将一个类的属性映射到多张表中。

```
<join
        table="tablename"
        schema="owner"
        catalog="catalog"
        fetch="join|select"
        inverse="true|false"
        optional="true|false">

        <key ... />

        <property ... />
        ...
</join>
```

❶	**`table`**：被连接表的名称。
❷	`schema` (可选):覆盖由根 `<hibernate-mapping>` 元素指定的模式名称。
❸	`catalog` (可选): 覆盖由根 `<hibernate-mapping>` 元素指定的目录名称。
❹	`fetch` (可选 - 默认是 `join`): 如果设置为默认值 `join` ，Hibernate 将使用一个内连接来得到这个类或其超类定义的 `<join>` ，而使用一个外连接来得到其子类定义的 `<join>` 。如果设置为 `select` ，则 Hibernate 将为子类定义的 `<join>` 使用顺序选择。这仅在一行数据表示一个子类的对象的时候才会发生。对这个类和其超类定义的 `<join>` ，依然会使用内连接得到。
❺	`inverse` (可选 - 默认是 `false`): 如果打开，Hibernate 不会插入或者更新此连接定义的属性。
❻	`optional` (可选 - 默认是 `false`): 如果打开，Hibernate 只会在此连接定义的属性非空时插入一行数据，并且总是使用一个外连接来得到这些属性。

例如，一个人(person)的地址(address)信息可以被映射到单独的表中(并保留所有属性的值类型语义)：

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

此特性常常对遗留数据模型有用，我们推荐表个数比类个数少，以及细粒度的领域模型。然而，在单独的继承树上切换继承映射策略是有用的，后面会解释这点。

5.1.19. 键(key)

我们目前已经见到过 `<key>` 元素多次了。 这个元素在父映射元素定义了对新表的连接，并且在被连接表中定义了一个外键引用原表的主键的情况下经常使用。

```
<key
        column="columnname"
        on-delete="noaction|cascade"
        property-ref="propertyName"
        not-null="true|false"
        update="true|false"
        unique="true|false"
/>
```

❶	**`column` (可选):** 外键字段的名称。也可以通过嵌套的 **`<column>`** 指定。
❷	`on-delete` (可选, 默认是 `noaction`): 表明外键关联是否打开数据库级别的级联删除。
❸	`property-ref` (可选): 表明外键引用的字段不是原表的主键(提供给遗留数据)。
❹	`not-null` (可选): 表明外键的字段不可为空(这意味着无论何时外键都是主键的一部分)。
❺	`update` (可选): 表明外键决不应该被更新(这意味着无论何时外键都是主键的一部分)。
❻	`unique` (可选): 表明外键应有唯一性约束 (这意味着无论何时外键都是主键的一部分)。

对那些看重删除性能的系统，我们推荐所有的键都应该定义为 `on-delete="cascade"` ，这样 Hibernate 将使用数据库级的 `ON CASCADE DELETE` 约束，而不是多个 `DELETE` 语句。 注意，这个特性会绕过 Hibernate 通常对版本数据(versioned data)采用的乐观锁策略。

`not-null` 和 `update` 属性在映射单向一对多关联的时候有用。如果你映射一个单向一对多关联到非空的(non-nullable)外键，你必须用 `<key not-null="true">` 定义此键字段。

5.1.20. 字段和规则元素（column and formula elements）

任何接受 `column` 属性的映射元素都可以选择接受 `<column>` 子元素。同样的，`formula` 子元素也可以替换 `<formula>` 属性。

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"
        default="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

`column` 和 `formula` 属性甚至可以在同一个属性或关联映射中被合并来表达，例如，一些奇异的连接条件。

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
</many-to-one>
```

5.1.21. 引用(import)

假设你的应用程序有两个同样名字的持久化类，但是你不想在Hibernate查询中使用他们的全限定名。除了依赖 `auto-import="true"` 以外，类也可以被显式地“import(引用)”。你甚至可以引用没有被明确映射的类和接口。

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"
        rename="ShortName"
/>
```

❶	`class`: 任何Java类的全限定名。
❷	`rename` (可选 - 默认为类的全限定名): 在查询语句中可以使用的名字。

5.1.22. any

这是属性映射的又一种类型。 <any> 映射元素定义了一种从多个表到类的多态关联。这种类型的映射常常需要多于一个字段。第一个字段持有被关联实体的类型，其他的字段持有标识符。对这种类型的关联来说，不可能指定一个外键约束，所以这当然不是映射(多态)关联的通常的方式。你只应该在非常特殊的情况下使用它(比如，审计log，用户会话数据等等)。

`meta-type` 属性使得应用程序能指定一个将数据库字段的值映射到持久化类的自定义类型。这个持久化类包含有用 `id-type` 指定的标识符属性。 你必须指定从meta-type的值到类名的映射。

```
<any name="being" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="propertyName"
        id-type="idtypename"
        meta-type="metatypename"
        cascade="cascade_style"
        access="field|property|ClassName"
        optimistic-lock="true|false"
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

❶	`name`：属性名
❷	`id-type`：标识符类型
❸	`meta-type` (可选 -默认是 `string`): 允许辨别标志(discriminator)映射的任何类型
❹	`cascade` (可选 -默认是 `none`): 级联的类型
❺	`access` (可选 -默认是 `property`): Hibernate 用来访问属性值的策略。
❻	`optimistic-lock` (可选 -默认是 `true`): 表明更新此组件是否需要获取乐观锁。换句话说，当这个属性变脏时，是否增加版本号(Version)

5.2. Hibernate 的类型

5.2.1. 实体(Entities)和值(values)

为了理解很多与持久化服务相关的Java语言级对象的行为，我们需要把它们分为两类：

实体*entity* 独立于任何持有实体引用的对象。与通常的Java模型相比，不再被引用的对象会被当作垃圾收集掉。实体必须被显式的保存和删除(除非保存和删除是从父实体向子实体引发的级联)。这和ODMG模型中关于对象通过可触及保持持久性有一些不同——比较起来更加接近应用程序对象通常在一个大系统中的使用方法。实体支持循环引用和交叉引用，它们也可以加上版本信息。

一个实体的持久状态包含指向其他实体和值类型实例的引用。值可以是原始类型，集合(不是集合中的对象)，组件或者特定的不可变对象。与实体不同，值(特别是集合和组件)是通过可触及性来进行持久化和删除的。因为值对象(和原始类型数据)是随着包含他们的实体而被持久化和删除的，他们不能被独立的加上版本信息。值没有独立的标识，所以他们不能被两个实体或者集合共享。

直到现在，我们都一直使用术语“持久类”(persistent class)来代表实体。我们仍然会这么做。 然而严格说来，不是所有的用户自定义的，带有持久化状态的类都是实体。组件就是用户自定义类，却是值语义的。 `java.lang.String` 类型的java属性也是值语义的。给了这个定义以后，我们可以说所有JDK提供的类型(类)都是值类型的语义，而用于自定义类型可能被映射为实体类型或值类型语义。采用哪种类型的语义取决于开发人员。在领域模型中，寻找实体类的一个好线索是共享引用指向这个类的单一实例，而组合或聚合通常被转化为值类型。

我们会在本文档中重复碰到这两个概念。

挑战在于将java类型系统(和开发者定义的实体和值类型)映射到 SQL/数据库类型系统。Hibernate提供了连接两个系统之间的桥梁：对于实体类型，我们使用 `<class>` , `<subclass>` 等等。对于值类型，我们使用 `<property>` , `<component>` 及其他，通常跟随着 `type` 属性。这个属性的值是Hibernate 的映射类型的名字。Hibernate提供了许多现成的映射(标准的JDK值类型)。你也可以编写自己的映射类型并实现自定义的变换策略，随后我们会看到这点。

所有的Hibernate内建类型，除了collections以外，都支持空(null)语义。

5.2.2. 基本值类型

内建的 基本映射类型可以大致分为

`integer, long, short, float, double, character, byte, boolean, yes_n`

这些类型都对应Java的原始类型或者其封装类，来符合(特定厂商的)SQL 字段类型。 `boolean, yes_no` 和 `true_false` 都是Java 中 `boolean` 或者 `java.lang.Boolean` 的另外说法。

`string`

从 `java.lang.String` 到 `VARCHAR` (或者 Oracle的 `VARCHAR2`)的映射。

`date, time, timestamp`

从 `java.util.Date` 和其子类到SQL类型 `DATE` , `TIME` 和 `TIMESTAMP` (或等价类型)的映射。

`calendar, calendar_date`

从 `java.util.Calendar` 到SQL 类型 `TIMESTAMP` 和 `DATE` (或等价类型)的映射。

`big_decimal, big_integer`

从 `java.math.BigDecimal` 和 `java.math.BigInteger` 到 `NUMERIC` (或者 Oracle 的 `NUMBER` 类型)的映射。

`locale, timezone, currency`

从 `java.util.Locale` , `java.util.TimeZone` 和 `java.util.Currency` 到 `VARCHAR` (或者 Oracle 的 `VARCHAR2` 类型)的映射. `Locale` 和 `Currency` 的实例被映射为它们的ISO代码。 `TimeZone` 的实例被影射为它的 `ID` 。

`class`

从 `java.lang.Class` 到 `VARCHAR` (或者 Oracle 的 `VARCHAR2` 类型)的映射。 `Class` 被映射为它的全限定名。

`binary`

把字节数组(byte arrays)映射为对应的 SQL二进制类型。

`text`

把长Java字符串映射为SQL的 `CLOB` 或者 `TEXT` 类型。

`serializable`

把可序列化的Java类型映射到对应的SQL二进制类型。你也可以为一个并非默认为基本类型的可序列化Java类或者接口指定Hibernate类型 `serializable` 。

`clob, blob`

JDBC 类 `java.sql.Clob` 和 `java.sql.Blob` 的映射。某些程序可能不适合使用这个类型，因为blob和clob对象可能在一个事务之外是无法重用的。(而且, 驱动程序对这种类型的支持充满着补丁和前后矛盾。)

`imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date,`

一般来说，映射类型被假定为是可变的Java类型，只有对不可变Java类型，Hibernate会采取特定的优化措施，应用程序会把这些对象作为不可变对象处理。比如，你不应该对作为 `imm_timestamp` 映射的Date执行 `Date.setTime()` 。要改变属性的值，并且保存这一改变，应用程序必须对这一属性重新设置一个新的（不一样的）对象。

实体及其集合的唯一标识可以是除了 `binary` 、 `blob` 和 `clob` 之外的任何基础类型。(联合标识也是允许的，后面会说到。)

在 `org.hibernate.Hibernate` 中，定义了基础类型对应的 `Type` 常量。比如， `Hibernate.STRING` 代表 `string` 类型。

5.2.3. 自定义值类型

开发者创建属于他们自己的值类型也是很容易的。比如说，你可能希望持久化 `java.lang.BigInteger` 类型的属性，持久化成为 `VARCHAR` 字段。Hibernate没有内置这样一种类型。自定义类型能够映射一个属性(或集合元素)到不止一个数据库表字段。比如说，你可能有这样的Java属性： `getName()` / `setName()` ，这是 `java.lang.String` 类型的，对应的持久化到三个字段： `FIRST_NAME` , `INITIAL` , `SURNAME` 。

要实现一个自定义类型，可以实现 `org.hibernate.UserType` 或 `org.hibernate.CompositeUserType` 中的任一个，并且使用类型的Java全限定类名来定义属性。请查看 `org.hibernate.test.DoubleStringType` 这个例子，看看它是怎么做的。

```
<property name="twoStrings" type="org.hibernate.test.DoubleStrin
gType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

注意使用 <column> 标签来把一个属性映射到多个字段的做法。

`CompositeUserType` , `EnhancedUserType` , `UserCollectionType` ,和 `UserVersionType` 接口为更特殊的使用方式提供支持。

你甚至可以在一个映射文件中提供参数给一个 `UserType` 。 为了这样做，你的 `UserType` 必须实现 `org.hibernate.usertype.ParameterizedType` 接口。为了给自定义类型提供参数，你可以在映射文件中使用 <type> 元素。

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType"
>
        <param name="default">0</param>
    </type>
</property>
```

现在， `UserType` 可以从传入的 `Properties` 对象中得到 `default` 参数的值。

如果你非常频繁地使用某一 `UserType` ，可以为他定义一个简称。这可以通过使用 <typedef> 元素来实现。Typedefs为一自定义类型赋予一个名称，并且如果此类型是参数化的，还可以包含一系列默认的参数值。

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType"
 name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

也可以根据具体案例通过属性映射中的类型参数覆盖在typedef中提供的参数。

尽管 Hibernate 内建的丰富的类型和对组件的支持意味着你可能很少 需要使用自定义类型。不过，为那些在你的应用中经常出现的(非实体)类使用自定义类型也是一个好方法。例如，一个 `MonetaryAmount` 类使用 `CompositeUserType` 来映射是不错的选择，虽然他可以很容易地被映射成组件。这样做的动机之一是抽象。使用自定义类型，以后假若你改变表示金额的方法时，它可以保证映射文件不需要修改。

5.3. 多次映射同一个类

对特定的持久化类，映射多次是允许的。这种情形下，你必须指定*entity name*来区别不同映射实体的对象实例。（默认情况下，实体名字和类名是相同的。）Hibernate在操作持久化对象、编写查询条件，或者把关联映射到指定实体时，允许你指定这个entity name（实体名字）。

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

注意这里关联是如何用 `entity-name` 来代替 `class` 的。

5.4. SQL中引号包围的标识符

你可通过在映射文档中使用反向引号
( )把表名或者字段名包围起来，以强制Hibernate在生成的SQL中把标识符用引号包围起来
Dialect`(方言)来使用正确的引号风格(通常是双引号，但是在SQL Server中是括
号，MySQL中是反向引号)。

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned
"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.5. 其他元数据(Metadata)

XML 并不适用于所有人, 因此有其他定义Hibernate O/R 映射元数据(metadata)的方法。

5.5.1. 使用 XDoclet 标记

很多Hibernate使用者更喜欢使用XDoclet `@hibernate.tags` 将映射信息直接嵌入到源代码中。我们不会在本文档中涉及这个方法，因为严格说来，这属于XDoclet的一部分。然而，我们包含了如下使用XDoclet映射的 `Cat` 类的例子。

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mother;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /*
     * @hibernate.id
     *  generator-class="native"
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="PARENT_ID"
     */
    public Cat getMother() {
        return mother;
    }
    void setMother(Cat mother) {
        this.mother = mother;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
```

```
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  inverse="true"
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
```

```
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

参考Hibernate网站更多的Xdoclet和Hibernate的例子

5.5.2. 使用 JDK 5.0 的注解(Annotation)

JDK 5.0 在语言级别引入了 XDoclet 风格的标注，并且是类型安全的，在编译期进行检查。这一机制比XDoclet的注解更为强大，有更好的工具和IDE支持。例如，IntelliJ IDEA，支持JDK 5.0注解的自动完成和语法高亮 。EJB规范的新修订版(JSR-220)使用 JDK 5.0的注解作为entity beans的主要元数据(metadata)机制。Hibernate 3 实现了JSR-220 (the persistence API)的 `EntityManager` ，支持通过 *Hibernate Annotations*包定义映射元数据。这个包作为单独的部分下载，支持EJB3 (JSR-220)和Hibernate3的元数据。

这是一个被注解为EJB entity bean 的POJO类的例子

```
@Entity(access = AccessType.FIELD)
public class Customer implements Serializable {

    @Id;
    Long id;

    String firstName;
    String lastName;
    Date birthday;

    @Transient
    Integer age;

    @Embedded
    private Address homeAddress;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="CUSTOMER_ID")
    Set<Order> orders;

    // Getter/setter and business methods
}
```

注意：对 JDK 5.0 注解 (和 JSR-220)支持的工作仍然在进行中,并未完成。更多细节请参阅Hibernate Annotations 模块。

5.6. 数据库生成属性（Generated Properties）

Generated properties指的是其值由数据库生成的属性。一般来说，如果对象有任何属性由数据库生成值，Hibernate应用程序需要进行 `刷新(refresh)` 。但如果把属性标明为generated，就可以转由Hibernate来负责这个动作。实际上。对定义了generated properties的实体,每当Hibernate执行一条SQL INSERT或者UPDATE语句，会立刻执行一条select来获得生成的值。

被标明为generated的属性还必须是 non-insertable和 non-updateable的。只有第 5.1.7 节 “版本（version）(可选)”，第 5.1.8 节 “timestamp (可选)”和第 5.1.9 节 “property”可以被标明为generated。

`never` (默认) 标明此属性值不是从数据库中生成。

`insert` - 标明此属性值在insert的时候生成，但是不会在随后的update时重新生成。比如说创建日期就归属于这类。注意虽然第 5.1.7 节 “版本（version）(可选)”和第 5.1.8 节 “timestamp (可选)”属性可以被标注为generated，但是不适用这个选项...

`always` - 标明此属性值在insert和update时都会被生成。

5.7. 辅助数据库对象(Auxiliary Database Objects)

Allows CREATE and DROP of arbitrary database objects, in conjunction with Hibernate's schema evolution tools, to provide the ability to fully define a user schema within the Hibernate mapping files. Although designed specifically for creating and dropping things like triggers or stored procedures, really any SQL command that can be run via a `java.sql.Statement.execute()` method is valid here (ALTERs, INSERTS, etc). There are essentially two modes for defining auxiliary database objects... 帮助CREATE和DROP任意数据库对象，与Hibernate的schema交互工具组合起来，可以提供在Hibernate映射文件中完全定义用户schema的能力。虽然这是为创建和销毁trigger(触发器）或stored procedure(存储过程）等特别设计的，实际上任何可以在 `java.sql.Statement.execute()` 方法中执行的SQL命令都可以在此使用（比如ALTER, INSERT，等等）。本质上有两种模式来定义辅助数据库对象...

第一种模式是在映射文件中显式声明CREATE和DROP命令：

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

第二种模式是提供一个类，这个类知道如何组织CREATE和DROP命令。这个特别类必须实现 `org.hibernate.mapping.AuxiliaryDatabaseObject` 接口。

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
</hibernate-mapping>
```

还有，这些数据库对象可以特别指定为仅在特定的方言中才使用。

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9Dialec
t"/>
        <dialect-scope name="org.hibernate.dialect.OracleDialect
"/>
    </database-object>
</hibernate-mapping>
```

第 6 章 集合类(Collections)映射

目录

6.1. 持久化集合类(Persistent collections)

(译者注：在阅读本章的时候，以后整个手册的阅读过程中，我们都会面临一个名词方面的问题，那就是“集合”。"Collections"和"Set"在中文里对应都被翻译为“集合”，但是他们的含义很不一样。Collections是一个超集，Set是其中的一种。大部分情况下，本译稿中泛指的未加英文注明的“集合”，都应当理解为“Collections”。在有些二者同时出现，可能造成混淆的地方，我们用“集合类”来特指“Collecions”,“集合(Set)”来指"Set"，一般都会在后面的括号中给出英文。希望大家在阅读时联系上下文理解，不要造成误解。 与此同时，“元素”一词对应的英文“element”，也有两个不同的含义。其一为集合的元素，是内存中的一个变量；另一含义则是XML文档中的一个标签所代表的元素。也请注意区别。 本章中,特别是后半部分是需要反复阅读才能理解清楚的。如果遇到任何疑问,请记住,英文版本的reference是惟一标准的参考资料。)

Hibernate要求持久化集合值字段必须声明为接口，比如：

```
public class Product {
    private String serialNumber;
    private Set parts = new HashSet();

    public Set getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }
}
```

实际的接口可能是 `java.util.Set` , `java.util.Collection` , `java.util.List` , `java.util.Map` , `java.util.SortedSet` , `java.util.SortedMap` 或者...任何你喜欢的类型！("任何你喜欢的类型" 代表你需要编写 `org.hibernate.usertype.UserCollectionType` 的实现.)

注意我们是如何用一个 `HashSet` 实例来初始化实例变量的.这是用于初始化新创建(尚未持久化)的类实例中集合值属性的最佳方法。当你持久化这个实例时——比如通过调用 `persist()` ——Hibernate 会自动把 `HashSet` 替换为Hibernate自己的 `Set` 实现。观察下面的错误：

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); //Okay, kittens collection is a Set
(HashSet) cat.getKittens(); //Error!
```

根据不同的接口类型，被Hibernate注射的持久化集合类的表现类似`HashMap`，`HashSet`，`TreeMap`，`TreeSet` or `ArrayList`。

集合类实例具有值类型的通常行为。当被持久化对象引用后，他们会自动被持久化，当不再被引用后，自动被删除。假若实例被从一个持久化对象传递到另一个，它的元素可能从一个表转移到另一个表。两个实体不能共享同一个集合类实例的引用。因为底层关系数据库模型的原因，集合值属性无法支持空值语义；Hibernate对空的集合引用和空集合不加区别。

你不需要过多的为此担心。就如同你平时使用普通的Java集合类一样来使用持久化集合类。只是要确认你理解了双向关联的语义（后文讨论）。

6.2. 集合映射（ Collection mappings ）

用于映射集合类的Hibernate映射元素取决于接口的类型。比如， `<set>` 元素用来映射 `Set` 类型的属性。

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
    </set>
</class>
```

除了 `<set>` ,还有 `<list>` , `<map>` , `<bag>` , `<array>` 和 `<primitive-array>` 映射元素。 `<map>` 具有代表性：

```
<map
    name="propertyName"
    table="table_name"
    schema="schema_name"
    lazy="true|extra|false"
    inverse="true|false"
    cascade="all|none|save-update|delete|all-delete-orphan|delet
e-orphan"
    sort="unsorted|natural|comparatorClass"
    order-by="column_name asc|desc"
    where="arbitrary sql where condition"
    fetch="join|select|subselect"
    batch-size="N"
    access="field|property|ClassName"
    optimistic-lock="true|false"
    mutable="true|false"
    node="element-name|."
    embed-xml="true|false"
>

    <key .... />
    <map-key .... />
    <element .... />
</map>
```

❶	**name** 集合属性的名称
❷	`table` （可选——默认为属性的名称）这个集合表的名称(不能在一对多的关联关系中使用)
❸	`schema` (可选) 表的schema的名称, 他将覆盖在根元素中定义的schema
❹	`lazy` (可选--默认为true) 可以用来关闭延迟加载(false)，指定一直使用预先抓取,或者打开"extra-lazy" 抓取，此时大多数操作不会初始化集合类(适用于非常大的集合)
❺	`inverse` (可选——默认为 `false`) 标记这个集合作为双向关联关系中的方向一端。
❻	`cascade` (可选——默认为 `none`) 让操作级联到子实体
❼	`sort` (可选)指定集合的排序顺序, 其可以为自然的(`natural`)或者给定一个用来比较的类。
❽	`order-by` (可选, 仅用于jdk1.4) 指定表的字段(一个或几个)再加上asc或者desc(可选), 定义Map,Set和Bag的迭代顺序
❾	`where` (可选) 指定任意的SQL where条件, 该条件将在重新载入或者删除这个集合时使用(当集合中的数据仅仅是所有可用数据的一个子集时这个条件非常有用)
❿	`fetch` (可选, 默认为 `select`) 用于在外连接抓取、通过后续select抓取和通过后续subselect抓取之间选择。
⓫	`batch-size` (可选, 默认为 `1`) 指定通过延迟加载取得集合实例的批处理块大小（"batch size"）。
⓬	`access` (可选-默认为属性property):Hibernate取得集合属性值时使用的策略
⓭	乐观锁 (可选 - 默认为 `true`): 对集合的状态的改变会是否导致其所属的实体的版本增长。(对一对多关联来说，关闭这个属性常常是有理的)
⓮	`mutable(可变）` (可选 - 默认为 `true`): 若值为 `false` ,表明集合中的元素不会改变（在某些情况下可以进行一些小的性能优化）。

6.2.1. 集合外键(Collection foreign keys)

集合实例在数据库中依靠持有集合的实体的外键加以辨别。此外键作为集合关键字段（*collection key column*）（或多个字段）加以引用。集合关键字段通过 `<key>` 元素映射。

在外键字段上可能具有非空约束。对于大多数集合来说，这是隐含的。对单向一对多关联来说，外键字段默认是可以为空的，因此你可能需要指明 `not-null="true"` 。

```
<key column="productSerialNumber" not-null="true"/>
```

外键约束可以使用 `ON DELETE CASCADE` 。

```
<key column="productSerialNumber" on-delete="cascade"/>
```

对 `<key>` 元素的完整定义，请参阅前面的章节。

6.2.2. 集合元素（Collection elements）

集合几乎可以包含任何其他的Hibernate类型，包括所有的基本类型、自定义类型、组件，当然还有对其他实体的引用。存在一个重要的区别：位于集合中的对象可能是根据“值”语义来操作（其声明周期完全依赖于集合持有者），或者它可能是指向另一个实体的引用，具有其自己的生命周期。在后者的情况下，被作为集合持有的状态考虑的，只有两个对象之间的“连接”。

被包容的类型被称为集合元素类型（*collection element type*）。集合元素通过 `<element>` 或 `<composite-element>` 映射，或在其是实体引用的时候，通过 `<one-to-many>` 或 `<many-to-many>` 映射。前两种用于使用值语义映射元素，后两种用于映射实体关联。

6.2.3. 索引集合类(Indexed collections)

所有的集合映射，除了set和bag语义的以外，都需要指定一个集合表的索引字段 *(index column)*——用于对应到数组索引，或者 `List` 的索引，或者 `Map` 的关键字。通过 <map-key> ，`Map` 的索引可以是任何基础类型；若通过 <map-key-many-to-many> ，它也可以是一个实体引用；若通过 <composite-map-key> ，它还可以是一个组合类型。数组或列表的索引必须是 `integer` 类型，并且使用 <list-index> 元素定义映射。被映射的字段包含有顺序排列的整数（默认从0开始）。

```
<map-key
        column="column_name"
        formula="any SQL expression"
        type="type_name"
        node="@attribute-name"
        length="N"/>
```

❶	**`column` (可选):**保存集合索引值的字段名。
❷	`formula` (可选): 用于计算map关键字的SQL公式
❸	`type` (必须):映射键(map key)的类型。

```
<map-key-many-to-many
        column="column_name"
        formula="any SQL expression"
        class="ClassName"
/>
```

❶	**`column` (可选):**集合索引值中外键字段的名称
❷	`formula` (可选): 用于计算map关键字的外键的SQL公式
❸	`class` (必需):映射的键(map key)使用的实体类。

假若你的表没有一个索引字段,当你仍然希望使用 `List` 作为属性类型,你应该把此属性映射为Hibernate *<bag>*。从数据库中获取的时候，bag不维护其顺序，但也可选择性的进行排序。

从集合类可以产生很大一部分映射，覆盖了很多常见的关系模型。我们建议你试验schema生成工具，来体会一下不同的映射声明是如何被翻译为数据库表的。

6.2.4. 值集合于多对多关联(Collections of values and many-to-many associations)

任何值集合或者多对多关联需要专用的具有一个或多个外键字段的*collection table*、一个或多个*collection element column*，以及还可能有一个或多个索引字段。

对于一个值集合, 我们使用 <element> 标签。

```
<element
        column="column_name"
        formula="any SQL expression"
        type="typename"
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="element-name"
/>
```

❶	**column (可选):** 保存集合元素值的字段名。
❷	formula (可选): 用于计算元素的SQL公式
❸	type (必需):集合元素的类型

多对多关联*(many-to-many association)* 使用 <many-to-many> 元素定义.

```
<many-to-many
        column="column_name"
        formula="any SQL expression"
        class="ClassName"
        fetch="select|join"
        unique="true|false"
        not-found="ignore|exception"
        entity-name="EntityName"
        property-ref="propertyNameFromAssociatedClass"
        node="element-name"
        embed-xml="true|false"
    />
```

❶	**column (可选): 这个元素的外键关键字段名**
❷	formula (可选): 用于计算元素外键值的SQL公式.
❸	class (必需): 关联类的名称
❸	outer-join (可选 - 默认为 auto): 在Hibernate系统参数中 hibernate.use_outer_join 被打开的情况下,该参数用来允许使用outer join来载入此集合的数据。
❹	为此关联打开外连接抓取或者后续select抓取。这是特殊情况；对于一个实体及其指向其他实体的多对多关联进全预先抓取（使用一条单独的 SELECT ），你不仅需要对集合自身打开 join ，也需要对 <many-to-many> 这个内嵌元素打开此属性。
❺	对外键字段允许DDL生成的时候生成一个惟一约束。这使关联变成了一个高效的一对多关联。（此句存疑：原文为This makes the association multiplicity effectively one to many.)
❻	not-found (可选 - 默认为 exception): 指明引用的外键中缺少某些行该如何处理： ignore 会把缺失的行作为一个空引用处理。
❼	entity-name (可选): 被关联的类的实体名，作为 class 的替代。
❽	property-ref：(可选) 被关联到此外键(foreign key)的类中的对应属性的名字。若未指定，使用被关联类的主键。

例子：首先, 一组字符串：

```
<set name="names" table="NAMES">
    <key column="GROUPID"/>
    <element column="NAME" type="string"/>
</set>
```

包含一组整数的bag(还设置了 order-by 参数指定了迭代的顺序)：

```
<bag name="sizes"
        table="item_sizes"
        order-by="size asc">
    <key column="item_id"/>
    <element column="size" type="integer"/>
</bag>
```

一个实体数组,在这个案例中是一个多对多的关联(注意这里的实体是自动管理生命周期的对象（lifecycle objects）, cascade="all"):

```
<array name="addresses"
        table="PersonAddress"
        cascade="persist">
    <key column="personId"/>
    <list-index column="sortOrder"/>
    <many-to-many column="addressId" class="Address"/>
</array>
```

一个map,通过字符串的索引来指明日期：

```
<map name="holidays"
        table="holidays"
        schema="dbo"
        order-by="hol_name asc">
    <key column="id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

一个组件的列表：（下一章讨论）

```
<list name="carComponents"
        table="CarComponents">
    <key column="carId"/>
    <list-index column="sortOrder"/>
    <composite-element class="CarComponent">
        <property name="price"/>
        <property name="type"/>
        <property name="serialNumber" column="serialNum"/>
    </composite-element>
</list>
```

6.2.5. 一对多关联（One-to-many Associations）

一对多关联__通过外键连接两个类对应的表,而没有中间集合表。 这个关系模型失去了一些Java集合的语义:

- 一个被包含的实体的实例只能被包含在一个集合的实例中
- 一个被包含的实体的实例只能对应于集合索引的一个值中

一个从 `Product` 到 `Part` 的关联需要关键字字段,可能还有一个索引字段指向 `Part` 所对应的表。 <one-to-many> 标记指明了一个一对多的关联。

```
<one-to-many
        class="ClassName"
        not-found="ignore|exception"
        entity-name="EntityName"
        node="element-name"
        embed-xml="true|false"
    />
```

❶	`class` **(**必须**):**被关联类的名称。
❷	`not-found` (可选 - 默认为 `exception`): 指明若缓存的标示值关联的行缺失,该如何处理: `ignore` 会把缺失的行作为一个空关联处理。
❸	`entity-name` (可选): 被关联的类的实体名，作为 `class` 的替代。

例子

```
<set name="bars">
    <key column="foo_id"/>
    <one-to-many class="org.hibernate.Bar"/>
</set>
```

注意: <one-to-many> 元素不需要定义任何字段。 也不需要指定表名。

重要提示:如果 一对多 关联中的外键字段定义成 `NOT NULL` ,你必须把 <key> 映射声明为 `not-null="true"` ,或者使用双向关联，并且标明 `inverse="true"` 。参阅本章后面关于双向关联的讨论。

下面的例子展示一个 `Part` 实体的map,把name作为关键字。(`partName` 是 `Part` 的持久化属性)。注意其中的基于公式的索引的用法。

```
<map name="parts"
        cascade="all">
    <key column="productId" not-null="true"/>
    <map-key formula="partName"/>
    <one-to-many class="Part"/>
</map>
```

6.3. 高级集合映射（Advanced collection mappings）

6.3.1. 有序集合（Sorted collections）

Hibernate支持实现 `java.util.SortedMap` 和 `java.util.SortedSet` 的集合。你必须在映射文件中指定一个比较器：

```
<set name="aliases"
            table="person_aliases"
            sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

`sort` 属性中允许的值包括 `unsorted` , `natural` 和某个实现了 `java.util.Comparator` 的类的名称。

分类集合的行为事实上象 `java.util.TreeSet` 或者 `java.util.TreeMap` 。

如果你希望数据库自己对集合元素排序，可以利用 `set` , `bag` 或者 `map` 映射中的 `order-by` 属性。这个解决方案只能在jdk1.4或者更高的jdk版本中才可以实现(通过LinkedHashSet或者 LinkedHashMap实现)。 它是在SQL查询中完成排序，而不是在内存中。

```
<set name="aliases" table="person_aliases" order-by="lower(name)
 asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

注意: 这个 `order-by` 属性的值是一个SQL排序子句而不是HQL的！

关联还可以在运行时使用集合 `filter()` 根据任意的条件来排序。

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.n
ame" ).list();
```

6.3.2. 双向关联（Bidirectional associations）

双向关联允许通过关联的任一端访问另外一端。在Hibernate中, 支持两种类型的双向关联:

一对多（one-to-many）

Set或者bag值在一端, 单独值(非集合)在另外一端

多对多（many-to-many）

两端都是set或bag值

要建立一个双向的多对多关联，只需要映射两个many-to-many关联到同一个数据库表中，并再定义其中的一端为*inverse*(使用哪一端要根据你的选择，但它不能是一个索引集合)。

这里有一个many-to-many的双向关联的例子;每一个category都可以有很多items,每一个items可以属于很多categories：

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="CATEGORY_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

如果只对关联的反向端进行了改变，这个改变不会被持久化。 这表示Hibernate为每个双向关联在内存中存在两次表现,一个从A连接到B,另一个从B连接到A。如果你回想一下Java对象模型，我们是如何在Java中创建多对多关系的，这可以让你更容易理解：

```
category.getItems().add(item);          // The category now "kno
ws" about the relationship
item.getCategories().add(category);     // The item now "knows"
about the relationship

session.persist(item);                   // The relationship won
''t be saved!
session.persist(category);               // The relationship wil
l be saved
```

非反向端用于把内存中的表示保存到数据库中。

要建立一个一对多的双向关联，你可以通过把一个一对多关联，作为一个多对一关联映射到到同一张表的字段上，并且在"多"的那一端定义 `inverse="true"` 。

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <set name="children" inverse="true">
        <key column="parent_id"/>
        <one-to-many class="Child"/>
    </set>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

在“一”这一端定义 `inverse="true"` 不会影响级联操作，二者是正交的概念！

6.3.3. 双向关联，涉及有序集合类

对于有一端是 `<list>` 或者 `<map>` 的双向关联，需要加以特别考虑。假若子类中的一个属性映射到索引字段，没问题，我们仍然可以在集合类映射上使用 `inverse="true"` ：

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <property name="name"
        not-null="true"/>
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

但是，假若子类中没有这样的属性存在，我们不能认为这个关联是真正的双向关联（信息不对称，在关联的一端有一些另外一端没有的信息）。在这种情况下，我们不能使用 `inverse="true"` 。我们需要这样用：

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>
```

注意在这个映射中，关联中集合类"值"一端负责来更新外键.TODO: Does this really result in some unnecessary update statements?

6.3.4. 三重关联（Ternary associations）

有三种可能的途径来映射一个三重关联。第一种是使用一个 `Map` ，把一个关联作为其索引：

```
<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/
>
    <one-to-many class="Contract"/>
</map>
```

```
<map name="connections">
    <key column="incoming_node_id"/>
    <map-key-many-to-many column="outgoing_node_id" class="Node"
/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

第二种方法是简单的把关联重新建模为一个实体类。这使我们最经常使用的方法。

最后一种选择是使用复合元素，我们会在后面讨论

6.3.5. 使用<idbag>

如果你完全信奉我们对于“联合主键（composite keys）是个坏东西”，和“实体应该使用（无机的）自己生成的代用标识符（surrogate keys）”的观点，也许你会感到有一些奇怪，我们目前为止展示的多对多关联和值集合都是映射成为带有联合主键的表的！现在，这一点非常值得争辩；看上去一个单纯的关联表并不能从代用标识符中获得什么好处（虽然使用组合值的集合可能会获得一点好处）。不过，Hibernate提供了一个（一点点试验性质的）功能，让你把多对多关联和值集合应得到一个使用代用标识符的表去。

<idbag> 属性让你使用bag语义来映射一个 List (或 Collection)。

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
```

你可以理解， <idbag> 人工的id生成器，就好像是实体类一样！集合的每一行都有一个不同的人造关键字。但是，Hibernate没有提供任何机制来让你取得某个特定行的人造关键字。

注意 <idbag> 的更新性能要比普通的 <bag> 高得多！Hibernate可以有效的定位到不同的行，分别进行更新或删除工作，就如同处理一个list, map或者set一样。

在目前的实现中，还不支持使用 identity 标识符生成器策略来生成 <idbag> 集合的标识符。

6.4. 集合例子（Collection example）

在前面的几个章节的确非常令人迷惑。 因此让我们来看一个例子。这个类：

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=child
ren; }

    ....
    ....
}
```

这个类有一个 `Child` 的实例集合。如果每一个子实例至多有一个父实例, 那么最自然的映射是一个one-to-many的关联关系：

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

在以下的表定义中反应了这个映射关系：

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varcha
r(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references
 parent
```

如果父亲是必须的, 那么就可以使用双向one-to-many的关联了：

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="Parent" column="parent
_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

请注意 `NOT NULL` 的约束:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references
 parent
```

另外，如果你绝对坚持这个关联应该是单向的，你可以对 `<key>` 映射声明 `NOT NULL` 约束：

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

另外一方面,如果一个子实例可能有多个父实例, 那么就应该使用many-to-many关联：

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

表定义：

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varcha
r(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) refe
rences parent
alter table childset add constraint childsetfk1 (child_id) refer
ences child
```

更多的例子,以及一个完整的父/子关系映射的排练,请参阅第 21 章 示例：父子关系 *(Parent Child Relationships)*.

甚至可能出现更加复杂的关联映射,我们会在下一章中列出所有可能性。

第 7 章 关联关系映射

目录

7.1. 介绍

关联关系映射通常情况是最难配置正确的。在这个部分中，我们从单向关系映射开始，然后考虑双向关系映射，由浅至深讲述一遍典型的案例。在所有的例子中，我们都使用 `Person` 和 `Address` 。

我们根据映射关系是否涉及连接表以及多样性来划分关联类型。

在传统的数据建模中，允许为Null值的外键被认为是一种不好的实践，因此我们所有的例子中都使用不允许为Null的外键。这并不是Hibernate的要求，即使你删除掉不允许为Null的约束，Hibernate映射一样可以工作的很好。

7.2. 单向关联（Unidirectional associations）

7.2.1. 多对一(many to one)

单向*many-to-one*关联是最常见的单向关联关系。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addr
essId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.2.2. 一对一（one to one）

基于外键关联的单向一对一关联和单向多对一关联几乎是一样的。唯一的不同就是单向一对一关联中的外键字段具有唯一性约束。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addr
essId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

基于主键关联的单向一对一关联通常使用一个特定的id生成器。（请注意，在这个例子中我们掉换了关联的方向。）

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.2.3. 一对多（one to many）

基于外键关联的单向一对多关联是一种很少见的情况，并不推荐使用。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, pe
rsonId bigint not null )
```

我们认为对于这种关联关系最好使用连接表。

7.3. 使用连接表的单向关联（Unidirectional associations with join tables）

7.3.1. 一对多(one to many)

基于连接表的单向一对多关联 应该优先被采用。请注意，通过指定 `unique="true"` ，我们可以把多样性从多对多改变为一对多。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint
 not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.3.2. 多对一（many to one）

基于连接表的单向多对一关联在关联关系可选的情况下应用也很普遍。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary ke
y, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.3.3. 一对一（one to one）

基于连接表的单向一对一关联非常少见，但也是可行的。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary ke
y, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

7.3.4. 多对多（many to many）

最后，还有 单向多对多关联.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId
 bigint not null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

7.4. 双向关联（Bidirectional associations）

7.4.1. 一对多（one to many) / 多对一（many to one）

双向多对一关联 是最常见的关联关系。（这也是标准的父/子关联关系。）

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key, addr
essId bigint not null )
create table Address ( addressId bigint not null primary key )
```

如果你使用 `List` (或者其他有序集合类)，你需要设置外键对应的 `key` 列为 `not null` ,让Hibernate来从集合端管理关联，维护每个元素的索引（通过设置 `update="false"` and `insert="false"` 来对另一端反向操作）。

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class>
```

假若集合映射的 <key> 元素对应的底层外键字段是 NOT NULL 的，那么为这一key元素定义 not-null="true" 是很重要的。不要仅仅为可能的嵌套 <column> 元素定义 not-null="true" ， <key> 元素也是需要的。

7.4.2. 一对一（one to one）

基于外键关联的双向一对一关联也很常见。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class>
```

```
create table Person ( personId bigint not null primary key, addr
essId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

基于主键关联的一对一关联需要使用特定的id生成器。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.5. 使用连接表的双向关联（Bidirectional associations with join tables）

7.5.1. 一对多（one to many）/多对一（ many to one）

基于连接表的双向一对多关联。注意 `inverse="true"` 可以出现在关联的任意一端，即collection端或者join端。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId
 bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.5.2. 一对一（one to one）

基于连接表的双向一对一关联极为罕见，但也是可行的。

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary ke
y, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

7.5.3. 多对多（many to many）

最后，还有 双向多对多关联.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"  table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true"  table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId
 bigint not null, primary key (personId, addressId) )
create table Address ( addressId bigint not null primary key )
```

7.6. 更复杂的关联映射

更复杂的关联连接极为罕见。 通过在映射文档中嵌入SQL片断，Hibernate也可以处理更为复杂的情况。比如，假若包含历史帐户数据的表定义了 `accountNumber` , `effectiveEndDate` 和 `effectiveStartDate` 字段，按照下面映射：

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"
/>
    <property name="currentAccount" type="boolean">
        <formula>case when effectiveEndDate is null then 1 else
0 end</formula>
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/
>
```

那么我们可以对目前*(current)*实例(其 `effectiveEndDate` 为null)使用这样的关联映射:

```
<many-to-one name="currentAccountInfo"
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
    <formula>'1'</formula>
</many-to-one>
```

更复杂的例子,假想 `Employee` 和 `Organization` 之间的关联是通过一个 `Employment` 中间表维护的,而中间表中填充了很多历史雇员数据。那"雇员的最新雇主"这个关联（最新雇主就是 `startDate` 最后的那个）可以这样映射：

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join>
```

使用这一功能时可以充满创意，但通常更加实用的是用HQL或条件查询来处理这些情形。

第 8 章 组件（Component）映射

目录

组件(Component)这个概念在Hibernate中几处不同的地方为了不同的目的被重复使用.

8.1. 依赖对象（Dependent objects）

组件(Component)是一个被包含的对象，在持久化的过程中，它被当作值类型，而并非一个实体的引用。在这篇文档中，组件这一术语指的是面向对象的合成概念（而并不是系统构架层次上的组件的概念）。举个例子, 你对人(Person)这个概念可以像下面这样来建模：

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

在持久化的过程中, `姓名(Name)` 可以作为 `人(Person)` 的一个组件。需要注意的是:你应该为 `姓名` 的持久化属性定义getter和setter方法,但是你不需要实现任何的接口或申明标识符字段。

以下是这个例子的Hibernate映射文件:

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute
 optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

人员(Person)表中将包括 `pid` , `birthday` , `initial` , `first` 和 `last` 等字段。

就像所有的值类型一样, 组件不支持共享引用。 换句话说，两个人可能重名，但是两个Person对象应该包含两个独立的Name对象，只不过这两个Name对象具有“同样”的值。 组件的值可以为空，其定义如下。 每当Hibernate重新加载一个包含组件

的对象,如果该组件的所有字段为空，Hibernate将假定整个组件为空。 在大多数情况下,这样假定应该是没有问题的。

组件的属性可以是任意一种Hibernate类型（包括集合, 多对多关联， 以及其它组件等等）。嵌套组件不应该被当作一种特殊的应用(Nested components should not be considered an exotic usage)。 Hibernate倾向于支持细致的(fine-grained)对象模型。

`<component>` 元素还允许有 `<parent>` 子元素，用来表明component类中的一个属性是指向包含它的实体的引用。

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date">
    <component name="Name" class="eg.Name" unique="true">
        <parent name="namedPerson"/> <!-- reference back to the
Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component&gt;
</class>
```

8.2. 在集合中出现的依赖对象 (Collections of dependent objects)

Hibernate支持组件的集合(例如: 一个元素是姓名(Name)这种类型的数组)。 你可以使用 `<composite-element>` 标签替代 `<element>` 标签来定义你的组件集合。

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/&gt;
    <composite-element class="eg.Name"> <!-- class attribute req
uired -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>;
    </composite-element>
</set>
```

注意，如果你定义的Set包含组合元素(composite-element)，正确地实现 `equals()` 和 `hashCode()` 是非常重要的。

组合元素可以包含组件，但是不能包含集合。如果你的组合元素自身包含组件, 你必须使用 `<nested-composite-element>` 标签。这是一个相当特殊的案例 - 在一个组件的集合里，那些组件本身又可以包含其他的组件。这个时候你就应该考虑一下使用one-to-many关联是否会更恰当。 尝试对这个组合元素重新建模为一个实体－但是需要注意的是，虽然Java模型和重新建模前是一样的，关系模型和持久性语义会有细微的变化。

请注意如果你使用 `<set>` 标签,一个组合元素的映射不支持可能为空的属性. 当删除对象时， Hibernate必须使用每一个字段的值来确定一条记录(在组合元素表中，没有单独的关键字段)， 如果有为null的字段，这样做就不可能了。你必须作出一个选择，要么在组合元素中使用不能为空的属性，要么选择使用 `<list>` , `<map>` , `<bag>` 或者 `<idbag>` 而不是 `<set>` 。

组合元素有个特别的用法是它可以包含一个 `<many-to-one>` 元素。类似这样的映射允许你将一个many-to-many关联表映射为组合元素的集合。(A mapping like this allows you to map extra columns of a many-to-many association table to the composite element class.) 接下来的的例子是从 `Order` 到 `Item` 的一个多对多的关联关系, 关联属性是 `purchaseDate` , `price` 和 `quantity` 。

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true
">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- clas
s attribute is optional -->
        </composite-element>
    </set>
</class>
```

当然，当你定义Item时，你无法引用这些purchase，因此你无法实现双向关联查询。记住组件是值类型，并且不允许共享引用。某一个特定的 `Purchase` 可以放在 `Order` 的集合中，但它不能同时被 `Item` 所引用。

其实组合元素的这个用法可以扩展到三重或多重关联:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true
">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails" class="eg.Purcha
se"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

在查询中，表达组合元素的语法和关联到其他实体的语法是一样的。

8.3. 组件作为Map的索引（Components as Map indices ）

`<composite-map-key>` 元素允许你映射一个组件类作为一个 `Map` 的key，前提是你必须正确的在这个类中重写了 `hashCode()` 和 `equals()` 方法。

8.4. 组件作为联合标识符(Components as composite identifiers)

你可以使用一个组件作为一个实体类的标识符。 你的组件类必须满足以下要求：

- 它必须实现 `java.io.Serializable` 接口
- 它必须重新实现 `equals()` 和 `hashCode()` 方法, 始终和组合关键字在数据库中的概念保持一致

注意：在*Hibernate3*中，第二个要求并非是*Hibernate*强制必须的。但最好这样做。

你不能使用一个 `IdentifierGenerator` 产生组合关键字。一个应用程序必须分配它自己的标识符。

使用 `<composite-id>` 标签(并且内嵌 `<key-property>` 元素)代替通常的 `<id>` 标签。比如, `OrderLine` 类具有一个主键，这个主键依赖于 `Order` 的(联合)主键。

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>

    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class>
```

现在，任何指向 `OrderLine` 的外键都是复合的。在你的映射文件中，必须为其他类也这样声明。例如，一个指向 `OrderLine` 的关联可能被这样映射：

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one>
```

（注意在各个地方 `<column>` 标签都是 `column` 属性的替代写法。）

指向 `OrderLine` 的 `多对多` 关联也使用联合外键:

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set>
```

在 `Order` 中, `OrderLine` 的集合则是这样:

```
<set name="orderLines" inverse="true">
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <one-to-many class="OrderLine"/>
</set>
```

(与通常一样, `<one-to-many>` 元素不声明任何列.)

假若 `OrderLine` 本身拥有一个集合,它也具有组合外键。

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key>   <!-- a collection inherits the composite key typ
e -->
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class>
```

8.5. 动态组件 （Dynamic components）

你甚至可以映射 `Map` 类型的属性：

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component>
```

从 `<dynamic-component>` 映射的语义上来讲，它和 `<component>` 是相同的。 这种映射类型的优点在于通过修改映射文件，就可以具有在部署时检测真实属性的能力。利用一个DOM解析器，也可以在程序运行时操作映射文件。 更好的是，你可以通过 `Configuration` 对象来访问（或者修改）Hibernate的运行时元模型。

第 9 章 继承映射(Inheritance Mappings)

目录

9.1. 三种策略

Hibernate支持三种基本的继承映射策略：

- 每个类分层结构一张表(table per class hierarchy)
- 每个子类一张表(table per subclass)
- 每个具体类一张表(table per concrete class)

此外，Hibernate还支持第四种稍有不同的多态映射策略：

- 隐式多态(implicit polymorphism)

对于同一个继承层次内的不同分支，可以采用不同的映射策略，然后用隐式多 态来完成跨越整个层次的多态。但是在同一个 <class> 根元素 下，Hibernate不支持混合了元素 <subclass> 、 <joined-subclass> 和 <union-subclass> 的映射。在同一个 <class> 元素下，可以混合使用 “每个类分层结构一张表”（table per hierarchy） 和“每个子类一张表”（table per subclass） 这两种映射策略，这是通过结合元素 <subclass> 和 <join> 来实现的（见后）。

在多个映射文件中，可以直接在 `hibernate-mapping` 根下定义 `subclass` ， `union-subclass` 和 `joined-subclass` 。也就是说，你可以仅加入一个新的映射文件来扩展类层次。你必须在subclass的映射中指明 `extends` 属性，给出一个之前定义的超类的名字。注意，在以前，这一功能对映射文件的顺序有严格的要求，从Hibernate 3开始，使用extends关键字的时侯，对映射文件的顺序不再有要求；但在每个映射文件里，超类必须在子类之前定义。

```
 <hibernate-mapping>
     <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
          <property name="name" type="string"/>
     </subclass>
 </hibernate-mapping>
```

9.1.1. 每个类分层结构一张表(Table per class hierarchy)

假设我们有接口`Payment`和它的几个实现类：`CreditCardPayment`，`CashPayment`，和`ChequePayment`。则“每个类分层结构一张表”(Table per class hierarchy)的映射代码如下所示：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CRED
IT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

采用这种策略只需要一张表即可。它有一个很大的限制：要求那些由子类定义的字段， 如`CCTYPE`，不能有`非空(NOT NULL)`约束。

9.1.2. 每个子类一张表(Table per subclass)

对于上例中的几个类而言，采用“每个子类一张表”的映射策略，代码如下所示：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYM
ENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT"
>
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

需要四张表。三个子类表通过主键关联到超类表(因而关系模型实际上是一对一关联)。

9.1.3. 每个子类一张表(Table per subclass)，使用辨别标志(Discriminator)

注意，对“每个子类一张表”的映射策略，Hibernate的实现不需要辨别字段，而其他的对象/关系映射工具使用了一种不同于Hibernate的实现方法，该方法要求在超类表中有一个类型辨别字段(type discriminator column)。Hibernate采用的方法更 难实现，但从关系（数据库）的角度来看，按理说它更正确。若你愿意使用带有辨别字 段的“每个子类一张表”的策略，你可以结合使用 `<subclass>` 与 `<join>` ，如下所示：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CRED
IT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class>
```

可选的声明 `fetch="select"` ，是用来告诉Hibernate，在查询超类时， 不要使用外部连接(outer join)来抓取子类 `ChequePayment` 的数据。

9.1.4. 混合使用“每个类分层结构一张表”和“每个子类一张表”

你甚至可以采取如下方法混和使用“每个类分层结构一张表”和“每个子类一张表”这两种策略：

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CRED
IT">
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

对上述任何一种映射策略而言，指向根类 `Payment` 的 关联是使用 `<many-to-one>` 进行映射的。

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/
>
```

9.1.5. 每个具体类一张表(Table per concrete class)

对于“每个具体类一张表”的映射策略，可以采用两种方法。第一种方法是使用 <union-subclass> 。

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYME
NT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class>
```

这里涉及三张与子类相关的表。每张表为对应类的所有属性（包括从超类继承的属性）定义相应字段。

这种方式的局限在于，如果一个属性在超类中做了映射，其字段名必须与所有子类表中定义的相同。(我们可能会在Hibernate的后续发布版本中放宽此限制。) 不允许在联合子类(union subclass)的继承层次中使用标识生成器策略(identity generator strategy), 实际上, 主键的种子(primary key seed)不得不为同一继承层次中的全部被联合子类所共用.

假若超类是抽象类，请使用 `abstract="true"` 。当然，假若它不是抽象的，需要一个额外的表（上面的例子中，默认是 `PAYMENT` ），来保存超类的实例。

9.1.6. Table per concrete class, using implicit polymorphism

另一种可供选择的方法是采用隐式多态：

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

注意，我们没有在任何地方明确的提及接口 `Payment` 。同时注意 `Payment` 的属性在每个子类中都进行了映射。如果你想避免重复， 可以考虑使用XML实体(例如：位于 `DOCTYPE` 声明内的 `[ <!ENTITY allproperties SYSTEM "allproperties.xml"> ]` 和映射中的 `&allproperties;`)。

这种方法的缺陷在于，在Hibernate执行多态查询时(polymorphic queries)无法生成带 `UNION` 的SQL语句。

对于这种映射策略而言，通常用 `<any>` 来实现到 `Payment` 的多态关联映射。

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

9.1.7. 隐式多态和其他继承映射混合使用

对这一映射还有一点需要注意。因为每个子类都在各自独立的元素 `<class>` 中映射(并且 `Payment` 只是一个接口)，每个子类可以很容易的成为另一 个继承体系中的一部分！(你仍然可以对接口 `Payment` 使用多态查询。)

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"
/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN"
>
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT"
>
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

我们还是没有明确的提到 `Payment` 。 如果我们针对接口 `Payment` 执行查询 ——如 `from Payment` —— Hibernate 自动返回 `CreditCardPayment` (和它的子类，因为 它们也实现了接口 `Payment`)、 `CashPayment` 和 `Chequepayment` 的实例， 但不返回 `NonelectronicTransaction` 的实例。

9.2. 限制

对“每个具体类映射一张表”（table per concrete-class）的映射策略而言，隐式多态的 方式有一定的限制。而 <union-subclass> 映射的限制则没有那 么严格。

下面表格中列出了在Hibernte中“每个具体类一张表”的策略和隐式多态的限制。

表 9.1. 继承映射特性(Features of inheritance mappings)

继承策略 (Inheritance strategy)	多态多对一	多态一对一	多
每个类分层结构一张表	<many-to-one>	<one-to-one>	<or
每个子类一张表	<many-to-one>	<one-to-one>	<or
每个具体类一张表 (union-subclass)	<many-to-one>	<one-to-one>	<or (仅对于 inve 情况)
每个具体类一张表(隐式多态)	<any>	不支持	不支持

第 10 章 与对象共享

目录

Hibernate是完整的对象/关系映射解决方案，它提供了对象状态管理*(state management)*的功能，使开发者不再需要理会底层数据库系统的细节。 也就是说，相对于常见的JDBC/SQL持久层方案中需要`管理SQL语句`，Hibernate采用了更自然的面向对象的视角来持久化Java应用中的数据。

换句话说，使用Hibernate的开发者应该总是关注对象的状态*(state)*，不必考虑SQL语句的执行。 这部分细节已经由Hibernate掌管妥当，只有开发者在进行系统性能调优的时候才需要进行了解。

10.1. Hibernate对象状态(object states)

Hibernate定义并支持下列对象状态(state):

- 瞬时*(Transient)* - 由 `new` 操作符创建，且尚未与Hibernate `Session` 关联的对象被认定为瞬时(Transient)的。瞬时(Transient)对象不会被持久化到数据库中，也不会被赋予持久化标识(identifier)。 如果瞬时(Transient)对象在程序中没有被引用，它会被垃圾回收器(garbage collector)销毁。 使用Hibernate `Session` 可以将其变为持久(Persistent)状态。(Hibernate会自动执行必要的SQL语句)
- 持久*(Persistent)* - 持久(Persistent)的实例在数据库中有对应的记录，并拥有一个持久化标识(identifier)。 持久(Persistent)的实例可能是刚被保存的，或刚被加载的，无论哪一种，按定义，它存在于相关联的 `Session` 作用范围内。 Hibernate会检测到处于持久(Persistent)状态的对象的任何改动，在当前操作单元(unit of work)执行完毕时将对象数据(state)与数据库同步(synchronize)。 开发者不需要手动执行 `UPDATE` 。将对象从持久(Persistent)状态变成瞬时(Transient)状态同样也不需要手动执行 `DELETE` 语句。
- 脱管*(Detached)* - 与持久(Persistent)对象关联的 `Session` 被关闭后，对象就变为脱管(Detached)的。 对脱管(Detached)对象的引用依然有效，对象可继续被修改。脱管(Detached)对象如果重新关联到某个新的 `Session` 上， 会再次转变为持久(Persistent)的(在Detached其间的改动将被持久化到数据库)。 这个功能使得一种编程模型，即中间会给用户思考时间(user think-time)的长时间运行的操作单元(unit of work)的编程模型成为可能。 我们称之为应用程序事务，即从用户观点看是一个操作单元(unit of work)。

接下来我们来细致的讨论下状态(states)及状态间的转换(state transitions)（以及触发状态转换的Hibernate方法）。

10.2. 使对象持久化

Hibernate认为持久化类(persistent class)新实例化的对象是瞬时*(Transient)*的。 我们可通过将瞬时(Transient)对象与session关联而把它变为持久*(Persistent)*的。

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

如果 `Cat` 的持久化标识(identifier)是 `generated` 类型的， 那么该标识(identifier)会自动在 `save()` 被调用时产生并分配给 `cat` 。 如果 `Cat` 的持久化标识(identifier)是 `assigned` 类型的，或是一个复合主键(composite key)， 那么该标识(identifier)应当在调用 `save()` 之前手动赋予给 `cat` 。 你也可以按照EJB3 early draft中定义的语义，使用 `persist()` 替代 `save()` 。

此外，你可以用一个重载版本的 `save()` 方法。

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

如果你持久化的对象有关联的对象(associated objects)（例如上例中的 `kittens` 集合） 那么对这些对象（译注：pk和kittens）进行持久化的顺序是任意的（也就是说可以先对kittens进行持久化也可以先对pk进行持久化）， 除非你在外键列上有 `NOT NULL` 约束。 Hibernate不会违反外键约束，但是如果你用错误的顺序持久化对象（译注：在pk持久化之前持久化kitten），那么可能会违反 `NOT NULL` 约束。

通常你不会为这些细节烦心，因为你很可能会使用Hibernate的 传播性持久化*(transitive persistence)*功能自动保存相关联那些对象。 这样连违反 `NOT NULL` 约束的情况都不会出现了 - Hibernate会管好所有的事情。 传播性持久化(transitive persistence)将在本章稍后讨论。

10.3. 装载对象

如果你知道某个实例的持久化标识(identifier)，你就可以使用 `Session` 的 `load()` 方法 来获取它。 `load()` 的另一个参数是指定类的.class对象。 本方法会创建指定类的持久化实例，并从数据库加载其数据(state)。

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new
 Long(id) );
```

此外, 你可以把数据(state)加载到指定的对象实例上（覆盖掉该实例原来的数据）。

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

请注意如果没有匹配的数据库记录， `load()` 方法可能抛出无法恢复的异常(unrecoverable exception)。 如果类的映射使用了代理(proxy)， `load()` 方法会返回一个未初始化的代理，直到你调用该代理的某方法时才会去访问数据库。 若你希望在某对象中创建一个指向另一个对象的关联，又不想在从数据库中装载该对象时同时装载相关联的那个对象，那么这种操作方式就用得上的了。 如果为相应类映射关系设置了 `batch-size` ， 那么使用这种操作方式允许多个对象被一批装载（因为返回的是代理，无需从数据库中抓取所有对象的数据）。

如果你不确定是否有匹配的行存在，应该使用 `get()` 方法，它会立刻访问数据库，如果没有对应的记录，会返回null。

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

你甚至可以选用某个 `LockMode` ，用SQL的 `SELECT ... FOR UPDATE` 装载对象。 请查阅API文档以获取更多信息。

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

注意，任何关联的对象或者包含的集合都不会被以 `FOR UPDATE` 方式返回， 除非你指定了 `lock` 或者 `all` 作为关联(association)的级联风格(cascade style)。

任何时候都可以使用 `refresh()` 方法强迫装载对象和它的集合。如果你使用数据库触发器功能来处理对象的某些属性，这个方法就很有用了。

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger execut
es)
```

此处通常会出现一个重要问题: Hibernate会从数据库中装载多少东西？会执行多少条相应的SQL `SELECT` 语句？ 这取决于抓取策略*(fetching strategy)*，会在第 19.1 节 “ 抓取策略(Fetching strategies) ”中解释。

10.4. 查询

如果不知道所要寻找的对象的持久化标识，那么你需要使用查询。Hibernate支持强大且易于使用的面向对象查询语言(HQL)。 如果希望通过编程的方式创建查询，Hibernate提供了完善的按条件(Query By Criteria, QBC)以及按样例(Query By Example, QBE)进行查询的功能。 你也可以用原生SQL(native SQL)描述查询，Hibernate额外提供了将结果集(result set)转化为对象的支持。

10.4.1. 执行查询

HQL和原生SQL(native SQL)查询要通过为 `org.hibernate.Query` 的实例来表达。 这个接口提供了参数绑定、结果集处理以及运行实际查询的方法。 你总是可以通过当前 `Session` 获取一个 `Query` 对象：

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother whe
re cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kit
tens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

一个查询通常在调用 `list()` 时被执行，执行结果会完全装载进内存中的一个集合(collection)。 查询返回的对象处于持久(persistent)状态。如果你知道的查询只会返回一个对象，可使用 `list()` 的快捷方式 `uniqueResult()` 。 注意，使用集合预先抓取的查询往往会返回多次根对象（他们的集合类都被初始化了）。你可以通过一个集合来过滤这些重复对象。

10.4.1.1. 迭代式获取结果(Iterating results)

某些情况下，你可以使用 `iterate()` 方法得到更好的性能。 这通常是你预期返回的结果在session，或二级缓存(second-level cache)中已经存在时的情况。 如若不然， `iterate()` 会比 `list()` 慢，而且可能简单查询也需要进行多次数据库访问： `iterate()` 会首先使用*1*条语句得到所有对象的持久化标识(identifiers)，再根据持久化标识执行*n*条附加的select语句实例化实际的对象。

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likel
iness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

10.4.1.2. 返回元组(tuples)的查询

（译注：元组(tuples)指一条结果行包含多个对象） Hibernate查询有时返回元组(tuples)，每个元组(tuples)以数组的形式返回:

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.m
other mother")
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten  = tuple[0];
    Cat mother  = tuple[1];
    ....
}
```

10.4.1.3. 标量(Scalar)结果

查询可在 `select` 从句中指定类的属性，甚至可以调用SQL统计(aggregate)函数。 属性或统计结果被认定为"标量(Scalar)"的结果（而不是持久(persistent state)的实体）。

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from C
at cat " +
        "group by cat.color")
        .list()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

10.4.1.4. 绑定参数

接口 `Query` 提供了对命名参数(named parameters)、JDBC风格的 `问号(?)` 参数进行绑定的方法。 不同于*JDBC*，*Hibernate*对参数从*0*开始计数。 命名参数(named parameters)在查询字符串中是形如 `:name` 的标识符。 命名参数(named parameters)的优点是:

- 命名参数(named parameters)与其在查询串中出现的顺序无关
- 它们可在同一查询串中多次出现
- 它们本身是自我说明的

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name
= :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name
= ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name
in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

10.4.1.5. 分页

如果你需要指定结果集的范围（希望返回的最大行数/或开始的行数），应该使用 `Query` 接口提供的方法:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Hibernate 知道如何将这个有限定条件的查询转换成你的数据库的原生SQL(native SQL)。

10.4.1.6. 可滚动遍历(Scrollable iteration)

如果你的JDBC驱动支持可滚动的 `ResuleSet` ， `Query` 接口可以使用 `ScrollableResults` ，允许你在查询结果中灵活游走。

```
Query q = sess.createQuery("select cat.name, cat from DomesticCa
t cat " +
                            "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list
of cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add(
cats.get(1) );

}
cats.close()
```

请注意，使用此功能需要保持数据库连接（以及游标(cursor)）处于一直打开状态。如果你需要断开连接使用分页功能，请使用 `setMaxResult()` / `setFirstResult()`

10.4.1.7. 外置命名查询(Externalizing named queries)

你可以在映射文件中定义命名查询(named queries)。（如果你的查询串中包含可能被解释为XML标记(markup)的字符，别忘了用 `CDATA` 包裹起来。）

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

参数绑定及执行以编程方式(programatically)完成：

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

请注意实际的程序代码与所用的查询语言无关，你也可在元数据中定义原生SQL(native SQL)查询， 或将原有的其他的查询语句放在配置文件中，这样就可以让Hibernate统一管理，达到迁移的目的。

也请注意在 `<hibernate-mapping>` 元素中声明的查询必须有一个全局唯一的名字,而在 `<class>` 元素中声明的查询自动具有全局名,是通过类的全名加以限定的。比如 `eg.Cat.ByNameAndMaximumWeight` 。

10.4.2. 过滤集合

集合过滤器*(filter)*是一种用于一个持久化集合或者数组的特殊的查询。查询字符串中可以使用 `"this"` 来引用集合中的当前元素。

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.c
lass) )
    .list()
);
```

返回的集合可以被认为是一个包(bag, 无顺序可重复的集合(collection))，它是所给集合的副本。 原来的集合不会被改动（这与“过滤器(filter)”的隐含的含义不符，不过与我们期待的行为一致）。

请注意过滤器(filter)并不需要 `from` 子句（当然需要的话它们也可以加上）。过滤器(filter)不限定于只能返回集合元素本身。

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue
")
    .list();
```

即使无条件的过滤器(filter)也是有意义的。例如，用于加载一个大集合的子集：

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

10.4.3. 条件查询(Criteria queries)

HQL极为强大，但是有些人希望能够动态的使用一种面向对象API创建查询，而非在他们的Java代码中嵌入字符串。对于那部分人来说，Hibernate提供了直观的 `Criteria` 查询API。

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

`Criteria` 以及相关的 `样例(Example)` API将会再第 15 章 *条件查询(Criteria Queries)*中详细讨论。

10.4.4. 使用原生SQL的查询

你可以使用 `createSQLQuery()` 方法，用SQL来描述查询，并由Hibernate将结果集转换成对象。 请注意，你可以在任何时候调用 `session.connection()` 来获得并使用JDBC `Connection` 对象。 如果你选择使用Hibernate的API, 你必须把SQL别名用大括号包围起来:

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.cla
ss}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

和Hibernate查询一样，SQL查询也可以包含命名参数和占位参数。 可以在第 16 章 *Native SQL查询*找到更多关于Hibernate中原生SQL(native SQL)的信息。

10.5. 修改持久对象

事务中的持久实例（就是通过 `session` 装载、保存、创建或者查询出的对象） 被应用程序操作所造成的任何修改都会在 `Session` 被刷出（*flushed*）的时候被持久化（本章后面会详细讨论）。 这里不需要调用某个特定的方法（比如 `update()` ，设计它的目的是不同的）将你的修改持久化。 所以最直接的更新一个对象的方法就是在 `Session` 处于打开状态时 `load()` 它，然后直接修改即可：

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(6
9) );
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and
persisted
```

有时这种程序模型效率低下，因为它在同一Session里需要一条SQL `SELECT` 语句（用于加载对象） 以及一条SQL `UPDATE` 语句(持久化更新的状态)。 为此Hibernate提供了另一种途径，使用脱管(detached)实例。

请注意*Hibernate*本身不提供直接执行 *`UPDATE`* 或 *`DELETE`* 语句的*API*。 *Hibernate*提供的是__状态管理*(state management)*服务，你不必考虑要使用的语句*(statements)*。 JDBC是出色的执行SQL语句的API，任何时候调用 `session.connection()` 你都可以得到一个JDBC `Connection` 对象。 此外，在联机事务处理(OLTP)程序中，大量操作(mass operations)与对象/关系映射的观点是相冲突的。 Hibernate的将来版本可能会提供专门的进行大量操作(mass operation)的功能。 参考第 13 章 批量处理（*Batch processing*），寻找一些可用的批量(batch)操作技巧。

10.6. 修改脱管(Detached)对象

很多程序需要在某个事务中获取对象，然后将对象发送到界面层去操作，最后在一个新的事务保存所做的修改。 在高并发访问的环境中使用这种方式，通常使用附带版本信息的数据来保证这些"长"工作单元之间的隔离。

Hibernate通过提供 `Session.update()` 或 `Session.merge()` 重新关联脱管实例的办法来支持这种模型。

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

如果具有 `catId` 持久化标识的 `Cat` 之前已经被 `另一Session(secondSession)` 装载了， 应用程序进行重关联操作(reattach)的时候会抛出一个异常。

如果你确定当前session没有包含与之具有相同持久化标识的持久实例，使用 `update()` 。 如果想随时合并你的的改动而不考虑session的状态，使用 `merge()` 。 换句话说，在一个新session中通常第一个调用的是 `update()` 方法，以便保证重新关联脱管(detached)对象的操作首先被执行。

如果希望相关联的脱管对象（通过引用"可到达"的脱管对象）的数据也要更新到数据库时（并且也仅仅在这种情况）， 可以对该相关联的脱管对象单独调用 `update()` 当然这些可以自动完成，即通过使用传播性持久化*(transitive persistence)*，请看第 10.11 节 "传播性持久化(transitive persistence)"。

`lock()` 方法也允许程序重新关联某个对象到一个新session上。不过，该脱管(detached)的对象必须是没有修改过的！

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassoci
ate:
sess.lock(pk, LockMode.UPGRADE);
```

请注意，`lock()` 可以搭配多种`LockMode` ，更多信息请阅读API文档以及关于事务处理(transaction handling)的章节。重新关联不是`lock()` 的唯一用途。

其他用于长时间工作单元的模型会在第 11.3 节 “乐观并发控制(Optimistic concurrency control)”中讨论。

10.7. 自动状态检测

Hibernate的用户曾要求一个既可自动分配新持久化标识(identifier)保存瞬时(transient)对象，又可更新/重新关联脱管(detached)实例的通用方法。`saveOrUpdate()` 方法实现了这个功能。

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);   // update existing state (cat
 has a non-null id)
secondSession.saveOrUpdate(mate);  // save the new instance (mat
e has a null id)
```

`saveOrUpdate()` 用途和语义可能会使新用户感到迷惑。 首先，只要你没有尝试在某个session中使用来自另一session的实例，你就应该不需要使用 `update()` ， `saveOrUpdate()` ，或 `merge()` 。有些程序从来不用这些方法。

通常下面的场景会使用 `update()` 或 `saveOrUpdate()` ：

- 程序在第一个session中加载对象
- 该对象被传递到表现层
- 对象发生了一些改动
- 该对象被返回到业务逻辑层
- 程序调用第二个session的 `update()` 方法持久这些改动

`saveOrUpdate()` 做下面的事:

- 如果对象已经在本session中持久化了，不做任何事
- 如果另一个与本session关联的对象拥有相同的持久化标识(identifier)，抛出一个异常
- 如果对象没有持久化标识(identifier)属性，对其调用 `save()`
- 如果对象的持久标识(identifier)表明其是一个新实例化的对象，对其调用 `save()`

- 如果对象是附带版本信息的（通过 `<version>` 或 `<timestamp>` ）并且版本属性的值表明其是一个新实例化的对象， `save()` 它。
- 否则 `update()` 这个对象

`merge()` 可非常不同:

- 如果session中存在相同持久化标识(identifier)的实例，用用户给出的对象的状态覆盖旧有的持久实例
- 如果session没有相应的持久实例，则尝试从数据库中加载，或创建新的持久化实例
- 最后返回该持久实例
- 用户给出的这个对象没有被关联到session上，它依旧是脱管的

10.8. 删除持久对象

使用 `Session.delete()` 会把对象的状态从数据库中移除。 当然，你的应用程序可能仍然持有一个指向已删除对象的引用。所以，最好这样理解： `delete()` 的用途是把一个持久实例变成瞬时(transient)实例。

```
sess.delete(cat);
```

你可以用你喜欢的任何顺序删除对象，不用担心外键约束冲突。当然，如果你搞错了顺序，还是有可能引发在外键字段定义的 `NOT NULL` 约束冲突。 例如你删除了父对象，但是忘记删除孩子们。

10.9. 在两个不同数据库间复制对象

偶尔会用到不重新生成持久化标识(identifier)，将持久实例以及其关联的实例持久到不同的数据库中的操作。

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

`ReplicationMode` 决定在和数据库中已存在记录由冲突时， `replicate()` 如何处理。

- `ReplicationMode.IGNORE` - 忽略它
- `ReplicationMode.OVERWRITE` - 覆盖相同的行
- `ReplicationMode.EXCEPTION` - 抛出异常
- `ReplicationMode.LATEST_VERSION` - 如果当前的版本较新，则覆盖，否则忽略

这个功能的用途包括使录入的数据在不同数据库中一致，产品升级时升级系统配置信息，回滚non-ACID事务中的修改等等。（译注，non-ACID，非ACID;ACID，Atomic，Consistent，Isolated and Durable的缩写）

10.10. Session刷出(flush)

每间隔一段时间，`Session` 会执行一些必需的SQL语句来把内存中的对象的状态同步到JDBC连接中。这个过程被称为刷出*(flush)*，默认会在下面的时间点执行：

- 在某些查询执行之前
- 在调用 `org.hibernate.Transaction.commit()` 的时候
- 在调用 `Session.flush()` 的时候

涉及的SQL语句会按照下面的顺序发出执行：

1. 所有对实体进行插入的语句，其顺序按照对象执行 `Session.save()` 的时间顺序
2. 所有对实体进行更新的语句
3. 所有进行集合删除的语句
4. 所有对集合元素进行删除，更新或者插入的语句
5. 所有进行集合插入的语句
6. 所有对实体进行删除的语句，其顺序按照对象执行 `Session.delete()` 的时间顺序

（有一个例外是，如果对象使用 `native` 方式来生成ID（持久化标识）的话，它们一执行save就会被插入。）

除非你明确地发出了 `flush()` 指令，关于Session何时会执行这些JDBC调用是完全无法保证的，只能保证它们执行的前后顺序。 当然，Hibernate保证， `Query.list(..)` 绝对不会返回已经失效的数据，也不会返回错误数据。

也可以改变默认的设置，来让刷出(flush)操作发生的不那么频繁。 `FlushMode` 类定义了三种不同的方式。 仅在提交时刷出(仅当Hibernate的 `Transaction` API被使用时有效)， 按照刚才说的方式刷出， 以及除非明确使用 `flush()` 否则从不刷出。 最后一种模式对于那些需要长时间保持 `Session` 为打开或者断线状态的长时间运行的工作单元很有用。 (参见 第 11.3.2 节 “扩展周期的session和自动版本化”).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return
stale state

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

刷出(flush)期间，可能会抛出异常。（例如一个DML操作违反了约束）异常处理涉及到对Hibernate事务性行为的理解，因此我们将在第 11 章 事务和并发中讨论。

10.11. 传播性持久化(transitive persistence)

对每一个对象都要执行保存，删除或重关联操作让人感觉有点麻烦，尤其是在处理许多彼此关联的对象的时候。 一个常见的例子是父子关系。考虑下面的例子:

如果一个父子关系中的子对象是值类型(value typed)（例如，地址或字符串的集合）的，他们的生命周期会依赖于父对象，可以享受方便的级联操作(Cascading)，不需要额外的动作。 父对象被保存时，这些值类型(value typed)子对象也将被保存；父对象被删除时，子对象也将被删除。 这对将一个子对象从集合中移除是同样有效：Hibernate会检测到，并且因为值类型(value typed)的对象不可能被其他对象引用，所以Hibernate会在数据库中删除这个子对象。

现在考虑同样的场景，不过父子对象都是实体(entities)类型，而非值类型(value typed)（例如，类别与个体，或母猫和小猫）。 实体有自己的生命期，允许共享对其的引用（因此从集合中移除一个实体，不意味着它可以被删除）， 并且实体到其他关联实体之间默认没有级联操作的设置。 Hibernate默认不实现所谓的可到达即持久化（*persistence by reachability*）的策略。

每个Hibernate session的基本操作 - 包括 `persist(), merge(), saveOrUpdate(), delete(), lock(), refresh(), evi` - 都有对应的级联风格(cascade style)。 这些级联风格(cascade style)风格分别命名为 `create, merge, save-update, delete, lock, refresh, evict, replicate` 。 如果你希望一个操作被顺着关联关系级联传播，你必须在映射文件中指出这一点。例如：

```
<one-to-one name="person" cascade="persist"/>
```

级联风格(cascade style)是可组合的:

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

你可以使用 `cascade="all"` 来指定全部操作都顺着关联关系级联(cascaded)。 默认值是 `cascade="none"` ，即任何操作都不会被级联(cascaded)。

注意有一个特殊的级联风格(cascade style) `delete-orphan` ，只应用于one-to-many关联，表明 `delete()` 操作 应该被应用于所有从关联中删除的对象。

建议:

- 通常在 `<many-to-one>` 或 `<many-to-many>` 关系中应用级联(cascade)没什么意义。 级联(cascade)通常在 `<one-to-one>` 和 `<one-to-many>` 关系中比较有用。

- 如果子对象的寿命限定在父亲对象的寿命之内，可通过指定 `cascade="all,delete-orphan"` 将其变为自动生命周期管理的对象 *(lifecycle object)*。
- 其他情况，你可根本不需要级联(cascade)。但是如果你认为你会经常在某个事务中同时用到父对象与子对象，并且你希望少打点儿字，可以考虑使用 `cascade="persist,merge,save-update"` 。

可以使用 `cascade="all"` 将一个关联关系（无论是对值对象的关联，或者对一个集合的关联）标记为父/子关系的关联。 这样对父对象进行save/update/delete操作就会导致子对象也进行save/update/delete操作。

此外，一个持久的父对象对子对象的浅引用(mere reference)会导致子对象被同步save/update。 不过，这个隐喻(metaphor)的说法并不完整。除非关联是 `<one-to-many>` 关联并且被标记为 `cascade="delete-orphan"` ， 否则父对象失去对某个子对象的引用不会导致该子对象被自动删除。 父子关系的级联(cascading)操作准确语义如下：

- 如果父对象被 `persist()` ，那么所有子对象也会被 `persist()`
- 如果父对象被 `merge()` ，那么所有子对象也会被 `merge()`
- 如果父对象被 `save()` ， `update()` 或 `saveOrUpdate()` ，那么所有子对象则会被 `saveOrUpdate()`
- 如果某个持久的父对象引用了瞬时(transient)或者脱管(detached)的子对象，那么子对象将会被 `saveOrUpdate()`
- 如果父对象被删除，那么所有子对象也会被 `delete()`
- 除非被标记为 `cascade="delete-orphan"` （删除“孤儿”模式，此时不被任何一个父对象引用的子对象会被删除）， 否则子对象失掉父对象对其的引用时，什么事也不会发生。 如果有特殊需要，应用程序可通过显式调用delete()删除子对象。

最后，注意操作的级联可能是在调用期*(call time)*或者写入期*(flush time)*作用到对象图上的。所有的操作，如果允许，都在操作被执行的时候级联到可触及的关联实体上。然而， `save-upate` 和 `delete-orphan` 是在 `Session` flush的时候才作用到所有可触及的被关联对象上的。

10.12. 使用元数据

Hibernate中有一个非常丰富的元级别(meta-level)的模型，含有所有的实体和值类型数据的元数据。 有时这个模型对应用程序本身也会非常有用。 比如说，应用程序可能在实现一种“智能”的深度拷贝算法时， 通过使用Hibernate的元数据来了解哪些对象应该被拷贝（比如，可变的值类型数据）， 那些不应该（不可变的值类型数据，也许还有某些被关联的实体）。

Hibernate提供了 `ClassMetadata` 接口， `CollectionMetadata` 接口和 `Type` 层次体系来访问元数据。 可以通过 `SessionFactory` 获取元数据接口的实例。

```
Cat fritz = ......;
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.clas
s);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or asso
ciations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].i
sCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

第 11 章 事务和并发

目录

Hibernate的事务和并发控制很容易掌握。Hibernate直接使用JDBC连接和JTA资源，不添加任何附加锁定 行为。我们强烈推荐你花点时间了解JDBC编程，ANSI SQL查询语言和你使用 的数据库系统的事务隔离规范。

Hibernate不锁定内存中的对象。你的应用程序会按照你的数据库事务的隔离级别规定的那样运作。幸亏有了 `Session` ，使得Hibernate通过标识符查找，和实体查询（不是返回标量值的报表查询）提供了可重复的读取（Repeatable reads）功能， `Session` 同时也是事务范围内的缓存（cache）。

除了对自动乐观并发控制提供版本管理，针对行级悲观锁定，Hibernate也提供了辅助的（较小的)API，它使用了 `SELECT FOR UPDATE` 的SQL语法。本章后面会讨论乐观并发控制和这个API。

我们从 `Configuration` 层、 `SessionFactory` 层, 和 `Session` 层开始讨论Hibernate的并行控制、数据库事务和应用 程序的长事务。

11.1. Session和事务范围(transaction scope)

`SessionFactory` 对象的创建代价很昂贵，它是线程安全的对象，它为所有的应用程序线程所共享。它只创建一次，通常是在应用程序启动的时候，由一个 `Configuraion` 的实例来创建。

`Session` 对象的创建代价比较小，是非线程安全的，对于单个请求，单个会话、单个的 工作单元而言，它只被使用一次，然后就丢弃。只有在需要的时候，一个 `Session` 对象 才会获取一个JDBC的 `Connection` （或一个 `Datasource` ）对象，因此假若不使用的时候它不消费任何资源。

此外我们还要考虑数据库事务。数据库事务应该尽可能的短，降低数据库中的锁争用。 数据库长事务会阻止你的应用程序扩展到高的并发负载。因此，假若在用户思考期间让数据库事务开着，直到整个工作单元完成才关闭这个事务，这绝不是一个好的设计。

一个操作单元(Unit of work)的范围是多大？单个的Hibernate `Session` 能跨越多个数据库事务吗？还是一个 `Session` 的作用范围对应一个数据库事务的范围？应该何时打开 `Session` ，何时关闭 `Session` ？，你又如何划分数据库事务的边界呢？

11.1.1. 操作单元(Unit of work)

首先，别用*session-per-operation*这种反模式了，也就是说，在单个线程中， 不要因为一次简单的数据库调用，就打开和关闭一次 `Session` ！数据库事务也是如此。 应用程序中的数据库调用是按照计划好的次序，分组为原子的操作单元。（注意，这也意味着，应用程 序中，在单个的SQL语句发送之后，自动事务提交(auto-commit)模式失效了。这种模式专门为SQL控制台操作设计的。 Hibernate禁止立即自动事务提交模式，或者期望应用服务器禁止立即自动事务提交模式。）数据库事务绝不是可有可无的，任何与数据库之间的通讯都必须在某个事务中进行，不管你是在读还是在写数据。对读数据而言，应该避免auto-commit行为，因为很多小的事务比一个清晰定义的工作单元性能差。后者也更容易维护和扩展。

在多用户的client/server应用程序中，最常用的模式是 每个请求一个会话*(session-per-request)*。 在这种模式下，来自客户端的请求被发送到服务器端（即Hibernate持久化层运行的地方），一 个新的Hibernate `Session` 被打开，并且执行这个操作单元中所有的数据库操作。 一旦操作完成（同时对客户端的响应也准备就绪），session被同步，然后关闭。你也可以使用单 个数据库事务来处理客户端请求，在你打开 `Session` 之后启动事务，在你关闭 `Session` 之前提交事务。会话和请求之间的关系是一对一的关系，这种模式对 于大多数应用程序来说是很棒的。

实现才是真正的挑战。Hibernate内置了对"当前session(current session)" 的管理，用于简化此模式。你要做的一切就是在服务器端要处理请求的时候，开启事务，在响应发送给客户之前结束事务。你可以用任何方式来完成这一操作，通常的方案有 `ServletFilter` ，在service方法中进行pointcut的AOP拦截器，或者proxy/interception容器。EJB容器是实现横切诸如EJB session bean上的事务分界，用CMT对事务进行声明等方面的标准手段。假若你决定使用编程式的事务分界，请参考本章后面讲到的Hibernate `Transaction` API，这对易用性和代码可移植性都有好处。

在任何时间，任何地方，你的应用代码可以通过简单的调用 `sessionFactory.getCurrentSession()` 来访问"当前session"，用于处理请求。你总是会得到当前数据库事务范围内的 `Session` 。在使用本地资源或JTA环境时，必须配置它，请参见第 2.5 节 “上下文相关的（Contextual）Session”。

有时，将 `Session` 和数据库事务的边界延伸到"展示层被渲染后"会带来便利。有些serlvet应用程序在对请求进行处理后，有个单独的渲染期，这种延伸对这种程序特别有用。假若你实现你自己的拦截器，把事务边界延伸到展示层渲染结束后非常容易。然而，假若你依赖有容器管理事务的EJB，这就不太容易了，因为事务会在EJB方法返回后结束，而那是在任何展示层渲染开始之前。请访问Hibernate网站和论坛，你可以找到*Open Session in View*这一模式的提示和示例。

11.1.2. 长对话

session-per-request模式不仅仅是一个可以用来设计操作单元的有用概念。很多业务处理都需 要一系列完整的与用户之间的交互，而这些用户是指对数据库有交叉访问的用户。在基于web的应用和企业 应用中，跨用户交互的数据库事务是无法接受的。考虑下面的例子：

- 在界面的第一屏，打开对话框，用户所看到的数据是被一个特定的 `Session` 和数据 库事务载入(load)的。用户可以随意修改对话框中的数据对象。
- 5分钟后，用户点击“保存”，期望所做出的修改被持久化；同时他也期望自己是唯一修改这个信息的人，不会出现 修改冲突。

从用户的角度来看，我们把这个操作单元称为长时间运行的对话（conversation），或者(or 应用事务,application transaction)。 在你的应用程序中，可以有很多种方法来实现它。

头一个幼稚的做法是，在用户思考的过程中，保持 `Session` 和数据库事务是打开的， 保持数据库锁定，以阻止并发修改，从而保证数据库事务隔离级别和原子操作。这种方式当然是一个反模式， 因为锁争用会导致应用程序无法扩展并发用户的数目。

很明显，我们必须使用多个数据库事务来实现这个对话。在这个例子中，维护业务处理的 事务隔离变成了应用程序层的部分责任。一个对话通常跨越多个数据库事务。如果仅仅只有一 个数据库事务（最后的那个事务）保存更新过的数据，而所有其他事务只是单纯的读取数据（例如在一 个跨越多个请求/响应周期的向导风格的对话框中），那么应用程序事务将保证其原子性。这种方式比听 起来还要容易实现，特别是当你使用了Hibernate的下述特性的时候：

- 自动版本化 - Hibernate能够自动进行乐观并发控制 ，如果在用户思考 的过程中发生并发修改，Hibernate能够自动检测到。一般我们只在对话结束时才检查。
- 脱管对象（Detached Objects）- 如果你决定采用前面已经讨论过的 *session-per-request*模式，所有载入的实例在用户思考的过程 中都处于与Session脱离的状态。Hibernate允许你把与Session脱离的对象重新关联到Session 上，并且对修改进行持久化，这种模式被称为 *session-per-request-with-detached-objects*。自动版本化被用来隔离并发修改。
- *Extended (or Long) Session* - Hibernate 的 `Session` 可以在数据库事务提交之后和底层的JDBC连接断开，当一个新的客户端请求到来的时候，它又重新连接上底层的 JDBC连接。这种模式被称之为*session-per-conversation*，这种情况可 能会造成不必要的Session和JDBC连接的重新关联。自动版本化被用来隔离并发修改, `Session` 通常不允许自动flush,而是明确flush。

session-per-request-with-detached-objects 和 *session-per-conversation* 各有优缺点，我们在本章后面乐观并发 控制那部分再进行讨论。

11.1.3. 关注对象标识(Considering object identity)

应用程序可能在两个不同的 `Session` 中并发访问同一持久化状态，但是， 一个持久化类的实例无法在两个 `Session` 中共享。因此有两种不同的标识语义：

数据库标识

`foo.getId().equals( bar.getId() )`

JVM 标识

`foo==bar`

对于那些关联到 特定 `Session` （也就是在单个 `Session` 的范围内）上的对象来说，这 两种标识的语义是等价的，与数据库标识对应的JVM标识是由Hibernate来保 证的。不过，当应用程序在两个不同的session中并发访问具有同一持久化标 识的业务对象实例的时候，这个业务对象的两个实例事实上是不相同的（从 JVM识别来看）。这种冲突可以通过在同步和提交的时候使用自动版本化和乐 观锁定方法来解决。

这种方式把关于并发的头疼问题留给了Hibernate和数据库；由于在单个线程内，操作单元中的对象识别不 需要代价昂贵的锁定或其他意义上的同步，因此它同时可以提供最好的可伸缩性。只要在单个线程只持有一个 `Session` ，应用程序就不需要同步任何业务对象。在 `Session` 的范围内，应用程序可以放心的使用 `==` 进行对象比较。

不过，应用程序在 `Session` 的外面使用 `==` 进行对象比较可能会 导致无法预期的结果。在一些无法预料的场合，例如，如果你把两个脱管对象实例放进同一个 `Set` 的时候，就可能发生。这两个对象实例可能有同一个数据库标识（也就是说， 他们代表了表的同一行数据），从JVM标识的定义上来说，对脱管的对象而言，Hibernate无法保证他们 的的JVM标识一致。开发人员必须覆盖持久化类的 `equals()` 方法和 `hashCode()` 方法，从而实现自定义的对象相等语义。警告：不要使用数据库标识 来实现对象相等，应该使用业务键值，由唯一的，通常不变的属性组成。当一个瞬时对象被持久化的时 候，它的数据库标识会发生改变。如果一个瞬时对象（通常也包括脱管对象实例）被放入一 个 `Set` ，改变它的hashcode会导致与这个 `Set` 的关系中断。虽 然业务键值的属性不象数据库主键那样稳定不变，但是你只需要保证在同一个 `Set` 中的对象属性的稳定性就足够了。请到Hibernate网站去寻求这个问题更多的详细的讨论。请注意，这不是一 个有关Hibernate的问题，而仅仅是一个关于Java对象标识和判等行为如何实现的问题。

11.1.4. 常见问题

决不要使用反模式*session-per-user-session*或者 *session-per-application*（当然，这个规定几乎没有例外）。请注意， 下述一些问题可能也会出现在我们推荐的模式中，在你作出某个设计决定之前，请务必理解该模式的应用前提。

- `Session` 对象是非线程安全的。如果一个 `Session` 实例允许共享的话，那些支持并发运行的东东，例如HTTP request，session beans,或者是 Swing workers，将会导致出现资源争用（race condition）。如果在 `HttpSession` 中有 Hibernate 的 `Session` 的话（稍后讨论），你应该考虑同步访问你的Http session。 否则，只要用户足够快的点击浏览器的“刷新”，就会导致两个并发运行线程使用同一个 `Session` 。

- 一个由Hibernate抛出的异常意味着你必须立即回滚数据库事务，并立即关闭 `Session` （稍后会展开讨论）。如果你的 `Session` 绑定到一个应用程序上，你必 须停止该应用程序。回滚数据库事务并不会把你的业务对象退回到事务启动时候的状态。这 意味着数据库状态和业务对象状态不同步。通常情况下，这不是什么问题，因为异常是不可 恢复的,你必须在回滚之后重新开始执行。

- `Session` 缓存了处于持久化状态的每个对象（Hibernate会监视和检查脏数据）。 这意味着，如果你让 `Session` 打开很长一段时间，或是仅仅载入了过多的数据， `Session` 占用的内存会一直增长，直到抛出OutOfMemoryException异常。这个 问题的一个解决方法是调用 `clear()` 和 `evict()` 来管理 `Session` 的缓存，但是如果你需要大批量数据操作的话，最好考虑 使用存储过程。在第 13 章 批量处理（*Batch processing*）中有一些解决方案。在用户会话期间一直保持 `Session` 打开也意味着出现脏数据的可能性很高。

11.2. 数据库事务声明

数据库（或者系统）事务的声明总是必须的。在数据库事务之外，就无法和数据库通讯（这可能会让那些习惯于 自动提交事务模式的开发人员感到迷惑）。永远使用清晰的事务声明，即使只读操作也是如此。进行 显式的事务声明并不总是需要的，这取决于你的事务隔离级别和数据库的能力，但不管怎么说，声明事务总归有益无害。当然，一个单独的数据库事务总是比很多琐碎的事务性能更好，即时对读数据而言也是一样。

一个Hibernate应用程序可以运行在非托管环境中（也就是独立运行的应用程序，简单Web应用程序， 或者Swing图形桌面应用程序），也可以运行在托管的J2EE环境中。在一个非托管环境中，Hibernate 通常自己负责管理数据库连接池。应用程序开发人员必须手工设置事务声明，换句话说，就是手工启 动，提交，或者回滚数据库事务。一个托管的环境通常提供了容器管理事务(CMT)，例如事务装配通过可声 明的方式定义在EJB session beans的部署描述符中。可编程式事务声明不再需要，即使是 `Session` 的同步也可以自动完成。

让持久层具备可移植性是人们的理想,这种移植发生在非托管的本地资源环境，与依赖JTA但是使用BMT而非CMT的系统之间。在两种情况下你都可以使用编程式的事务管理。Hibernate提供了一套称为 `Transaction` 的封装API， 用来把你的部署环境中的本地事务管理系统转换到Hibernate事务上。这个API是可选的，但是我们强烈 推荐你使用，除非你用CMT session bean。

通常情况下，结束 `Session` 包含了四个不同的阶段:

- 同步session(flush,刷出到磁盘）
- 提交事务
- 关闭session
- 处理异常

session的同步(flush,刷出）前面已经讨论过了，我们现在进一步考察在托管和非托管环境下的事务声明和异常处理。

11.2.1. 非托管环境

如果Hibernat持久层运行在一个非托管环境中，数据库连接通常由Hibernate的简单（即非DataSource)连接池机制 来处理。session/transaction处理方式如下所示：

```
//Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

你不需要显式 `flush()` `Session` - 对 `commit()` 的调用会自动触发session的同步(取决于session的第 10.10 节 “Session刷出(flush)”)。调用 `close()` 标志session的结束。 `close()` 方法重要的暗示是， `session` 释放了JDBC连接。这段Java代码在非托管环境下和JTA环境下都可以运行。

更加灵活的方案是Hibernate内置的"current session"上下文管理，前文已经讲过：

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

你很可能从未在一个通常的应用程序的业务代码中见过这样的代码片断：致命的（系统）异常应该总是 在应用程序“顶层”被捕获。换句话说，执行Hibernate调用的代码（在持久层）和处理 `RuntimeException` 异常的代码（通常只能清理和退出应用程序）应该在不同 的应用程序逻辑层。Hibernate的当前上下文管理可以极大地简化这一设计，你所有的一切就是`SessionFactory` 。 异常处理将在本章稍后进行讨论。

请注意，你应该选择 `org.hibernate.transaction.JDBCTransactionFactory` (这是默认选项)，对第二个例子来说， `hibernate.current_session_context_class` 应该是`"thread"`

11.2.2. 使用JTA

如果你的持久层运行在一个应用服务器中（例如，在EJB session beans的后面），Hibernate获取 的每个数据源连接将自动成为全局JTA事务的一部分。 你可以安装一个独立的JTA实现，使用它而不使用EJB。Hibernate提供了两种策略进行JTA集成。

如果你使用bean管理事务（BMT），可以通过使用Hibernate的 `Transaction` API来告诉 应用服务器启动和结束BMT事务。因此，事务管理代码和在非托管环境下是一样的。

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

如果你希望使用与事务绑定的 `Session` ，也就是使用 `getCurrentSession()` 来简化上下文管理，你将不得不直接使用JTA `UserTransaction` API。

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction")
;

    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

在CMT方式下，事务声明是在session bean的部署描述符中，而不需要编程。 因此，代码被简化为:

```
// CMT idiom
Session sess = factory.getCurrentSession();

// do some work
...
```

在CMT/EJB中甚至会自动rollback，因为假若有未捕获的 `RuntimeException` 从session bean方法中抛出，这就会通知容器把全局事务回滚。这就意味着，在*BMT*或者*CMT*中，你根本就不需要使用*Hibernate* `Transaction` *API* ，你自动得到了绑定到事务的“当前”*Session*。

注意，当你配置Hibernate的transaction factory的时候，在直接使用JTA的时候（BMT），你应该选择 `org.hibernate.transaction.JTATransactionFactory` ,在CMT session bean中选择 `org.hibernate.transaction.CMTTransactionFactory` 。记得也要设置 `hibernate.transaction.manager_lookup_class` 。还有，确认你的 `hibernate.current_session_context_class` 未设置（为了向下兼容），或者设置为 `"jta"` 。

`getCurrentSession()` 在JTA环境中有一个弊端。对 `after_statement` 连接释放方式有一个警告，这是被默认使用的。因为JTA规范的一个很愚蠢的限制，Hibernate不可能自动清理任何未关闭的 `ScrollableResults` 或者 `Iterator` ，它们是由 `scroll()` 或 `iterate()` 产生的。你*must*通过在 `finally` 块中，显式

调用 `ScrollableResults.close()` 或者 `Hibernate.close(Iterator)` 方法来释放底层数据库游标。(当然，大部分程序完全可以很容易的避免在JTA或CMT代码中出现 `scroll()` 或 `iterate()` 。)

11.2.3. 异常处理

如果 `Session` 抛出异常 (包括任何 `SQLException`), 你应该立即回滚数据库事务，调用 `Session.close()` ，丢弃该 `Session` 实例。 `Session` 的某些方法可能会导致session 处于不一致的状态。所有由Hibernate抛出的异常都视为不可以恢复的。确保在 `finally` 代码块中调用 `close()` 方法，以关闭掉 `Session` 。

`HibernateException` 是一个非检查期异常（这不同于Hibernate老的版本）， 它封装了Hibernate持久层可能出现的大多数错误。我们的观点是，不应该强迫应用程序开发人员 在底层捕获无法恢复的异常。在大多数软件系统中，非检查期异常和致命异常都是在相应方法调用 的堆栈的顶层被处理的（也就是说，在软件上面的逻辑层），并且提供一个错误信息给应用软件的用户 （或者采取其他某些相应的操作）。请注意，Hibernate也有可能抛出其他并不属于 `HibernateException` 的非检查期异常。这些异常同样也是无法恢复的，应该 采取某些相应的操作去处理。

在和数据库进行交互时，Hibernate把捕获的 `SQLException` 封装为Hibernate的 `JDBCException` 。事实上，Hibernate尝试把异常转换为更有实际含义的 `JDBCException` 异常的子类。底层的 `SQLException` 可以 通过 `JDBCException.getCause()` 来得到。Hibernate通过使用关联到 `SessionFactory` 上的 `SQLExceptionConverter` 来 把 `SQLException` 转换为一个对应的 `JDBCException` 异常的子类。默认情况下， `SQLExceptionConverter` 可以通过配置dialect 选项指定；此外，也可以使用用户自定义的实现类（参考javadocs `SQLExceptionConverterFactory` 类来了解详情）。标准的 `JDBCException` 子类型是：

- `JDBCConnectionException` - 指明底层的JDBC通讯出现错误
- `SQLGrammarException` - 指明发送的SQL语句的语法或者格式错误
- `ConstraintViolationException` - 指明某种类型的约束违例错误
- `LockAcquisitionException` - 指明了在执行请求操作时，获取 所需的锁级别时出现的错误。
- `GenericJDBCException` - 不属于任何其他种类的原生异常

11.2.4. 事务超时

EJB这样的托管环境有一项极为重要的特性，而它从未在非托管环境中提供过，那就是事务超时。在出现错误的事务行为的时候，超时可以确保不会无限挂起资源、对用户没有交代。在托管(JTA)环境之外，Hibernate无法完全提供这一功能。但是，Hiberante至少可以控制数据访问，确保数据库级别的死锁，和返回巨大结果集的查询被限定在一个规定的时间内。在托管环境中，Hibernate会把事务超时转交给JTA。这一功能通过Hibernate `Transaction` 对象进行抽象。

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

注意 `setTimeout()` 不应该在CMT bean中调用，此时事务超时值应该是被声明式定义的。

11.3. 乐观并发控制(Optimistic concurrency control)

唯一能够同时保持高并发和高可伸缩性的方法就是使用带版本化的乐观并发控制。版本检查使用版本号、 或者时间戳来检测更新冲突（并且防止更新丢失）。Hibernate为使用乐观并发控制的代码提供了三种可 能的方法，应用程序在编写这些代码时，可以采用它们。我们已经在前面应用程序对话那部分展示了 乐观并发控制的应用场景，此外，在单个数据库事务范围内，版本检查也提供了防止更新丢失的好处。

11.3.1. 应用程序级别的版本检查(Application version checking)

未能充分利用Hibernate功能的实现代码中，每次和数据库交互都需要一个新的 `Session` ，而且开发人员必须在显示数据之前从数据库中重 新载入所有的持久化对象实例。这种方式迫使应用程序自己实现版本检查来确保 对话事务的隔离，从数据访问的角度来说是最低效的。这种使用方式和 entity EJB最相似。

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();

int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateExce
ption();
foo.setProperty("bar");

t.commit();
session.close();
```

`version` 属性使用 <version> 来映射，如果对象 是脏数据，在同步的时候，Hibernate会自动增加版本号。

当然，如果你的应用是在一个低数据并发环境下，并不需要版本检查的话，你照样可以使用 这种方式，只不过跳过版本检查就是了。在这种情况下，最晚提交生效（*last commit wins*）就是你的长对话的默认处理策略。 请记住这种策略可能会让应用软件的用户感到困惑，因为他们有可能会碰上更新丢失掉却没 有出错信息，或者需要合并更改冲突的情况。

很明显，手工进行版本检查只适合于某些软件规模非常小的应用场景，对于大多数软件应用场景 来说并不现实。通常情况下，不仅是单个对象实例需要进行版本检查，整个被修改过的关 联对象图也都需要进行版本检查。作为标准设计范例，Hibernate使用扩展周期的 `Session` 的方式，或者脱管对象实例的方式来提供自动版本检查。

11.3.2. 扩展周期的session和自动版本化

单个 `Session` 实例和它所关联的所有持久化对象实例都被用于整个 对话，这被称为*session-per-conversation*。Hibernate在同步的时候进行对象实例的版本检查，如果检测到并发修 改则抛出异常。由开发人员来决定是否需要捕获和处理这个异常（通常的抉择是给用户 提供一个合并更改，或者在无脏数据情况下重新进行业务对话的机会）。

在等待用户交互的时候， `Session` 断开底层的JDBC连接。这种方式 以数据库访问的角度来说是最高效的方式。应用程序不需要关心版本检查或脱管对象实例 的重新关联，在每个数据库事务中，应用程序也不需要载入读取对象实例。

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC
 connection, start transaction

foo.setProperty("bar");

session.flush();    // Only for last transaction in conversation
t.commit();         // Also return JDBC connection
session.close();    // Only for last transaction in conversation
```

`foo` 对象知道它是在哪个 `Session` 中被装入的。在一个旧session中开启一个新的数据库事务，会导致session获取一个新的连接，并恢复session的功能。将数据库事务提交，使得session从JDBC连接断开，并将此连接交还给连接池。在重新连接之后，要强制对你没有更新的数据进行一次版本检查，你可以对所有可能被其他事务修改过的对象，使用参数 `LockMode.READ` 来调用 `Session.lock()` 。你不用lock任何你正在更新的数据。一般你会在扩展的 `Session` 上设置 `FlushMode.NEVER` ，因此只有最后一个数据库事务循环才会真正的吧整个对话中发生的修改发送到数据库。因此，只有这最后一次数据库事务才会包含 `flush()` 操作，然后在整个对话结束后，还要 `close()` 这个session。

如果在用户思考的过程中， `Session` 因为太大了而不能保存，那么这种模式是有问题的。举例来说，一个 `HttpSession` 应该尽可能的小。由于 `Session` 是一级缓存，并且保持了所有被载入过的对象，因此 我们只应该在那些少量的request/response情况下使用这种策略。你应该只把一个 `Session` 用于单个对话，因为它很快就会出现脏数据。

（注意，早期的Hibernate版本需要明确的对 `Session` 进行disconnec和reconnect。这些方法现在已经过时了，打开事务和关闭事务会起到同样的效果。）

此外，也请注意，你应该让与数据库连接断开的 `Session` 对持久层保持 关闭状态。换句话说，在三层环境中，使用有状态的EJB session bean来持有 `Session` ， 而不要把它传递到web层（甚至把它序列化到一个单独的层），保存在 `HttpSession` 中。

扩展session模式，或者被称为每次对话一个*session(session-per-conversation)*, 在与自动管理当前session上下文联用的时候会更困难。你需要提供你自己的 `CurrentSessionContext` 实现。请参阅Hibernate Wiki以获得示例。

11.3.3. 脱管对象(deatched object)和自动版本化

这种方式下，与持久化存储的每次交互都发生在一个新的 `Session` 中。 然而，同一持久化对象实例可以在多次与数据库的交互中重用。应用程序操纵脱管对象实例的状态，这个脱管对象实例最初是在另一个 `Session` 中载入的，然后 调用 `Session.update()` ， `Session.saveOrUpdate()` , 或者 `Session.merge()` 来重新关联该对象实例。

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have be
en loaded already
t.commit();
session.close();
```

Hibernate会再一次在同步的时候检查对象实例的版本，如果发生更新冲突，就抛出异常。

如果你确信对象没有被修改过，你也可以调用 `lock()` 来设置 `LockMode.READ` （绕过所有的缓存，执行版本检查），从而取 代 `update()` 操作。

11.3.4. 定制自动版本化行为

对于特定的属性和集合，通过为它们设置映射属性 `optimistic-lock` 的值为 `false` ，来禁止Hibernate的版本自动增加。这样的话，如果该属性 脏数据，Hibernate将不再增加版本号。

遗留系统的数据库Schema通常是静态的，不可修改的。或者，其他应用程序也可能访问同一数据 库，根本无法得知如何处理版本号，甚至时间戳。在以上的所有场景中，实现版本化不能依靠 数据库表的某个特定列。在 <class> 的映射中设置 `optimistic-lock="all"` 可以在没有版本或者时间戳属性映射的情况下实现 版本检查，此时Hibernate将比较一行记录的每个字段的状态。请注意，只有当Hibernate能够比 较新旧状态的情况下，这种方式才能生效，也就是说， 你必须使用单个长生命周期 `Session` 模式，而不能使用 session-per-request-with-detached-objects模式。

有些情况下，只要更改不发生交错，并发修改也是允许的。当你在 <class> 的映射中设置 `optimistic-lock="dirty"` ，Hibernate在同步的时候将只比较有脏 数据的字段。

在以上所有场景中，不管是专门设置一个版本/时间戳列，还是进行全部字段/脏数据字段比较， Hibernate都会针对每个实体对象发送一条 `UPDATE` （带有相应的 `WHERE` 语句 ）的SQL语句来执行版本检查和数据更新。如果你对关联实体 设置级联关系使用传播性持久化（transitive persistence），那么Hibernate可能会执行不必 要的update语句。这通常不是个问题，但是数据库里面对*on update*点火 的触发器可能在脱管对象没有任何更改的情况下被触发。因此，你可以在 <class> 的映射中，通过设置 `select-before-update="true"` 来定制这一行为，强制Hibernate `SELECT` 这个对象实例，从而保证， 在更新记录之前，对象的确是被修改过。

11.4. 悲观锁定(Pessimistic Locking)

用户其实并不需要花很多精力去担心锁定策略的问题。通常情况下，只要为JDBC连接指定一下隔 离级别，然后让数据库去搞定一切就够了。然而，高级用户有时候希望进行一个排它的悲观锁定， 或者在一个新的事务启动的时候，重新进行锁定。

Hibernate总是使用数据库的锁定机制，从不在内存中锁定对象！

类 `LockMode` 定义了Hibernate所需的不同的锁定级别。一个锁定 可以通过以下的机制来设置:

- 当Hibernate更新或者插入一行记录的时候，锁定级别自动设置为 `LockMode.WRITE` 。
- 当用户显式的使用数据库支持的SQL格式 `SELECT ... FOR UPDATE` 发送SQL的时候，锁定级别设置为 `LockMode.UPGRADE`
- 当用户显式的使用Oracle数据库的SQL语句 `SELECT ... FOR UPDATE NOWAIT` 的时候，锁定级别设置 `LockMode.UPGRADE_NOWAIT`
- 当Hibernate在“可重复读”或者是“序列化”数据库隔离级别下读取数据的时候，锁定模式 自动设置为 `LockMode.READ` 。这种模式也可以通过用户显式指定进行设置。
- `LockMode.NONE` 代表无需锁定。在 `Transaction` 结束时， 所有的对象都切换到该模式上来。与session相关联的对象通过调用 `update()` 或者 `saveOrUpdate()` 脱离该模式。

"显式的用户指定"可以通过以下几种方式之一来表示:

- 调用 `Session.load()` 的时候指定 `锁定模式(LockMode)` 。
- 调用 `Session.lock()` 。
- 调用 `Query.setLockMode()` 。

如果在 `UPGRADE` 或者 `UPGRADE_NOWAIT` 锁定模式下调 用 `Session.load()` ，并且要读取的对象尚未被session载入过，那么对象 通过 `SELECT ... FOR UPDATE` 这样的SQL语句被载入。如果为一个对象调用 `load()` 方法时，该对象已经在另一个较少限制的锁定模式下被载入了，那 么Hibernate就对该对象调用 `lock()` 方法。

如果指定的锁定模式是 `READ` , `UPGRADE` 或 `UPGRADE_NOWAIT` ，那么 `Session.lock()` 就 执行版本号检查。（在 `UPGRADE` 或者 `UPGRADE_NOWAIT` 锁定模式下，执行 `SELECT ... FOR UPDATE` 这样的SQL语句。）

如果数据库不支持用户设置的锁定模式，Hibernate将使用适当的替代模式（而不是扔出异常）。 这一点可以确保应用程序的可移植性。

11.5. 连接释放模式(Connection Release Modes)

Hibernate关于JDBC连接管理的旧(2.x)行为是，`Session` 在第一次需要的时候获取一个连接，在session关闭之前一直会持有这个连接。Hibernate引入了连接释放的概念，来告诉session如何处理它的JDBC连接。注意，下面的讨论只适用于采用配置 `ConnectionProvider` 来提供连接的情况，用户自己提供的连接与这里的讨论无关。通过 `org.hibernate.ConnectionReleaseMode` 的不同枚举值来使用不用的释放模式:

- `ON_CLOSE` - 基本上就是上面提到的老式行为。Hibernate session在第一次需要进行JDBC操作的时候获取连接，然后持有它，直到session关闭。
- `AFTER_TRANSACTION` - 在 `org.hibernate.Transaction` 结束后释放连接。
- `AFTER_STATEMENT` (也被称做积极释放) - 在每一条语句被执行后就释放连接。但假若语句留下了与session相关的资源，那就不会被释放。目前唯一的这种情形就是使用 `org.hibernate.ScrollableResults` 。

`hibernate.connection.release_mode` 配置参数用来指定使用哪一种释放模式。可能的值有：

- `auto` (默认) - 这一选择把释放模式委派给 `org.hibernate.transaction.TransactionFactory.getDefaultRelease` 方法。对JTATransactionFactory来说，它会返回ConnectionReleaseMode.AFTER_STATEMENT;对JDBCTransactionFactory来说，则是ConnectionReleaseMode.AFTER_TRANSACTION。很少需要修改这一默认行为，因为假若设置不当，就会带来bug，或者给用户代码带来误导。
- `on_close` - 使用 ConnectionReleaseMode.ON_CLOSE. 这种方式是为了向下兼容的,但是已经完全不被鼓励使用了。
- `after_transaction` - 使用ConnectionReleaseMode.AFTER_TRANSACTION。这一设置不应该在JTA环境下使用。也要注意，使用ConnectionReleaseMode.AFTER_TRANSACTION的时候，假若session 处于auto-commit状态，连接会像AFTER_STATEMENT那样被释放。
- `after_statement` - 使用ConnectionReleaseMode.AFTER_STATEMENT。除此之外，会查询配置的 `ConnectionProvider` ，是否它支持这一设置（(`supportsAggressiveRelease()`)）。假若不支持，释放模式会被设置为ConnectionReleaseMode.AFTER_TRANSACTION。只有在你每次调用 `ConnectionProvider.getConnection()` 获取底层JDBC连接的时候，都可以确信获得同一个连接的时候，这一设置才是安全的；或者在auto-commit环境中，你可以不管是否每次都获得同一个连接的时候，这才是安全的。

第 12 章 拦截器与事件(Interceptors and events)

目录

应用程序能够响应Hibernate内部产生的特定事件是非常有用的。这样就允许实现某些通用的功能 以及允许对Hibernate功能进行扩展。

12.1. 拦截器(Interceptors)

`Interceptor` 接口提供了从会话(session)回调(callback)应用程序(application)的机制， 这种回调机制可以允许应用程序在持久化对象被保存、更新、删除或是加载之前，检查并（或）修改其 属性。一个可能的用途，就是用来跟踪审核(auditing)信息。例如：下面的这个 `拦截器` ，会在一个实现了 `Auditable` 接口的对象被创建时自动地设置 `createTimestamp` 属性，并在实现了 `Auditable` 接口的对象被更新时，同步更新 `lastUpdateTimestamp` 属性。

你可以直接实现 `Interceptor` 接口，也可以（最好）继承自 `EmptyInterceptor` 。

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames
[i] ) ) {
                    currentState[i] = new Date();
```

```
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i]
) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
            System.out.println("Creations: " + creates + ", Upda
tes: " + updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

拦截器可以有两种: `Session` 范围内的，和 `SessionFactory` 范围内的。

当使用某个重载的SessionFactory.openSession()使用 `Interceptor` 作为参数调用打开一个session的时候，就指定了 `Session` 范围内的拦截器。

```
Session session = sf.openSession( new AuditInterceptor() );
```

`SessionFactory` 范围内的拦截器要通过 `Configuration` 中注册，而这必须在创建 `SessionFactory` 之前。在这种情况下，给出的拦截器会被这个 `SessionFactory` 所打开的所有session使用了；除非session打开时明确指明了使用的拦截器。 `SessionFactory` 范围内的拦截器，必须是线程安全的，因为多个session可能并发使用这个拦截器，要因此小心不要保存与session相关的状态。

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

12.2. 事件系统(Event system)

如果需要响应持久层的某些特殊事件，你也可以使用Hibernate3的事件框架。 该事件系统可以用来替代拦截器，也可以作为拦截器的补充来使用。

基本上， `Session` 接口的每个方法都有相对应的事件。比如 `LoadEvent` ， `FlushEvent` ，等等（查阅XML配置文件 的DTD，以及 `org.hibernate.event` 包来获得所有已定义的事件的列表）。当某个方 法被调用时，Hibernate `Session` 会生成一个相对应的事件并激活所 有配置好的事件监听器。系统预设的监听器实现的处理过程就是被监听的方法要做的（被监听的方法所做的其实仅仅是激活监听器，“实际”的工作是由监听器完成的）。不过，你可以自由地选择实现 一个自己定制的监听器（比如，实现并注册用来处理处理 `LoadEvent` 的 `LoadEventListener` 接口）， 来负责处理所有的调用 `Session` 的 `load()` 方法的请求。

监听器应该被看作是单例(singleton)对象，也就是说，所有同类型的事件的处理共享同一个监听器实例，因此监听器 不应该保存任何状态（也就是不应该使用成员变量）。

用户定制的监听器应该实现与所要处理的事件相对应的接口，或者从一个合适的基类继承（甚至是从Hibernate自带的默认事件监听器类继承， 为了方便你这样做，这些类都被声明成non-final的了）。用户定制的监听器可以通过编程使用 `Configuration` 对象 来注册，也可以在Hibernate的XML格式的配置文件中进行声明（不支持在Properties格式的配置文件声明监听器）。 下面是一个用户定制的加载事件(load event)的监听器：

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListene
r interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadTy
pe loadType)
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(
), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

你还需要修改一处配置，来告诉Hibernate，除了默认的监听器，还要附加选定的监听器。

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener class="org.hibernate.event.def.DefaultLoad
EventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

看看用另一种方式，通过编程的方式来注册它。

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultL
oadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

通过在XML配置文件声明而注册的监听器不能共享实例。如果在多个 `<listener/>` 节点中使用 了相同的类的名字，则每一个引用都将会产生一个独立的实例。如果你需要在多个监听器类型之间共享 监听器的实例，则你必须使用编程的方式来进行注册。

为什么我们实现了特定监听器的接口，在注册的时候还要明确指出我们要注册哪个事件的监听器呢？ 这是因为一个类可能实现多个监听器的接口。在注册的时候明确指定要监听的事件，可以让启用或者禁用对某个事件的监听的配置工作简单些。

12.3. Hibernate的声明式安全机制

通常，Hibernate应用程序的声明式安全机制由会话外观层(session facade)所管理。 现在，Hibernate3允许某些特定的行为由JACC进行许可管理，由JAAS进行授权管理。 本功能是一个建立在事件框架之上的可选的功能。

首先，你必须要配置适当的事件监听器（event listener），来激活使用JAAS管理授权的功能。

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreD
eleteEventListener"/>
<listener type="pre-update" class="org.hibernate.secure.JACCPreU
pdateEventListener"/>
<listener type="pre-insert" class="org.hibernate.secure.JACCPreI
nsertEventListener"/>
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoa
dEventListener"/>
```

注意， `<listener type="..." class="..."/>` 只是 `<event type="..."><listener class="..."/></event&gt` 的简写，对每一个事件类型都必须严格的有一个监听器与之对应。

接下来，仍然在 `hibernate.cfg.xml` 文件中，绑定角色的权限：

```
<grant role="admin" entity-name="User" actions="insert,update,re
ad"/>
<grant role="su" entity-name="User" actions="*"/>
```

这些角色的名字就是你的JACC provider所定义的角色的名字。

第 13 章 批量处理（Batch processing）

目录

使用Hibernate将 100 000 条记录插入到数据库的一个很自然的做法可能是这样的

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

这段程序大概运行到 50 000 条记录左右会失败并抛出 `内存溢出异常（OutOfMemoryException）` 。 这是因为 Hibernate 把所有新插入的 `客户（Customer）` 实例在 session级别的缓存区进行了缓存的缘故。

我们会在本章告诉你如何避免此类问题。首先，如果你要执行批量处理并且想要达到一个理想的性能， 那么使用JDBC的批量（batching）功能是至关重要。将JDBC的批量抓取数量（batch size）参数设置到一个合适值 （比如，10-50之间）：

```
hibernate.jdbc.batch_size 20
```

注意,假若你使用了 `identiy` 标识符生成器,Hibernate在JDBC级别透明的关闭插入语句的批量执行。

你也可能想在执行批量处理时关闭二级缓存：

```
hibernate.cache.use_second_level_cache false
```

但是，这不是绝对必须的，因为我们可以显式设置 `CacheMode` 来关闭与二级缓存的交互。

13.1. 批量插入（Batch inserts）

如果要将很多对象持久化，你必须通过经常的调用 `flush()` 以及稍后调用 `clear()` 来控制第一级缓存的大小。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size //20,
与JDBC批量设置相同
        //flush a batch of inserts and release memory:
        //将本批插入的对象立即写入数据库并释放内存
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.2. 批量更新（Batch updates）

此方法同样适用于检索和更新数据。此外，在进行会返回很多行数据的查询时，你需要使用 `scroll()` 方法以便充分利用服务器端游标所带来的好处。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomer
s")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.3. StatelessSession (无状态session)接口

作为选择，Hibernate提供了基于命令的API，可以用detached object的形式把数据以流的方法加入到数据库，或从数据库输出。`StatelessSession` 没有持久化上下文，也不提供多少高层的生命周期语义。特别是，无状态session不实现第一级cache,也不和第二级缓存，或者查询缓存交互。它不实现事务化写，也不实现脏数据检查。用stateless session进行的操作甚至不级联到关联实例。stateless session忽略集合类(Collections)。通过stateless session进行的操作不触发Hibernate的事件模型和拦截器。无状态session对数据的混淆现象免疫，因为它没有第一级缓存。无状态session是低层的抽象，和低层JDBC相当接近。

```
StatelessSession session = sessionFactory.openStatelessSession()
;
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomer
s")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

注意在上面的例子中，查询返回的 `Customer` 实例立即被脱管(detach)。它们与任何持久化上下文都没有关系。

`StatelessSession` 接口定义的 `insert(), update()` 和 `delete()` 操作是直接的数据库行级别操作，其结果是立刻执行一条 `INSERT, UPDATE` 或 `DELETE` 语句。因此，它们的语义和 `Session` 接口定义的 `save(), saveOrUpdate()` 和 `delete()` 操作有很大的不同。

13.4. DML(数据操作语言)风格的操作(DML-style operations)

hence manipulating (using the SQL `Data Manipulation Language` (DML) statements: `INSERT`, `UPDATE`, `DELETE`) data directly in the database will not affect in-memory state. However, Hibernate provides methods for bulk SQL-style DML statement execution which are performed through the Hibernate Query Language (第 14 章 *HQL: Hibernate*查询语言). 就像已经讨论的那样，自动和透明的 对象/关系 映射（object/relational mapping）关注于管理对象的状态。 这就意味着对象的状态存在于内存，因此直接操作 (使用 SQL `Data Manipulation Language` (DML,数据操作语言）语句 ： `INSERT`, `UPDATE` 和 `DELETE`) 数据库中的数据将不会影响内存中的对象状态和对象数据。 不过，Hibernate提供通过Hibernate查询语言（第 14 章 *HQL: Hibernate*查询语言）来执行大批 量SQL风格的DML语句的方法。

`UPDATE` 和 `DELETE` 语句的语法为： `( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?` 有几点说明：

- 在FROM子句（from-clause）中，FROM关键字是可选的
- 在FROM子句（from-clause）中只能有一个实体名，它可以是别名。如果实体名是别名，那么任何被引用的属性都必须加上此别名的前缀；如果不是别名，那么任何有前缀的属性引用都是非法的。
- 不能在大批量HQL语句中使用第 14.4 节 "join 语法的形式"（显式或者隐式的都不行）。不过在WHERE子句中可以使用子查询。可以在where子句中使用子查询，子查询本身可以包含join。
- 整个WHERE子句是可选的。

举个例子，使用 `Query.executeUpdate()` 方法执行一个HQL `UPDATE` 语句(：(方法命名是来源于JDBC's `PreparedStatement.executeUpdate()`):

```
Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();

        String hqlUpdate = "update Customer c set c.name = :newN
ame where c.name = :oldName";
        // or String hqlUpdate = "update Customer set name = :ne
wName where name = :oldName";
        int updatedEntities = s.createQuery( hqlUpdate )
                .setString( "newName", newName )
                .setString( "oldName", oldName )
                .executeUpdate();
        tx.commit();
        session.close();
```

HQL `UPDATE` 语句，默认不会影响更新实体的第 5.1.7 节 “版本（version）(可选)”或者第 5.1.8 节 “timestamp (可选)”属性值。这和EJB3规范是一致的。但是，通过使用 `versioned update` ，你可以强制Hibernate正确的重置 `version` 或者 `timestamp` 属性值。这通过在 `UPDATE` 关键字后面增加 `VERSIONED` 关键字来实现的。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name
= :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

注意，自定义的版本类型(`org.hibernate.usertype.UserVersionType`)不允许和 `update versioned` 语句联用。

执行一个HQL `DELETE` ，同样使用 `Query.executeUpdate()` 方法:

```
Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();

        String hqlDelete = "delete Customer c where c.name = :ol
dName";
        // or String hqlDelete = "delete Customer where name = :
oldName";
        int deletedEntities = s.createQuery( hqlDelete )
                .setString( "oldName", oldName )
                .executeUpdate();
        tx.commit();
        session.close();
```

由 `Query.executeUpdate()` 方法返回的 `整型` 值表明了受此操作影响的记录数量。 注意这个数值可能与数据库中被（最后一条SQL语句）影响了的“行”数有关，也可能没有。一个大批量HQL操作可能导致多条实际的SQL语句被执行， 举个例子，对joined-subclass映射方式的类进行的此类操作。这个返回值代表了实际被语句影响了的记录数量。在那个joined-subclass的例子中， 对一个子类的删除实际上可能不仅仅会删除子类映射到的表而且会影响“根”表，还有可能影响与之有继承关系的joined-subclass映射方式的子类的表。

`INSERT` 语句的伪码是: `INSERT INTO EntityName properties_list select_statement`. 要注意的是:

- 只支持INSERT INTO ... SELECT ...形式,不支持INSERT INTO ... VALUES ...形式.

  properties_list和SQL `INSERT` 语句中的 `字段定义(column speficiation)` 类似。对参与继承树映射的实体而言，只有直接定义在给定的类级别的属性才能直接在properties_list中使用。超类的属性不被支持；子类的属性无意义。换句话说， `INSERT` 天生不支持多态。

- select*statement*可以是任何合法的*HQL*选择查询，不过要保证返回类型必须和要插入的类型完全匹配。目前，这一检查是在查询编译的时候进行的，而不是把它交给数据库。注意，在*Hibernate* `Type` 间如果只是等价（equivalent）而非相等(equal)_，会导致问题。定义为 `org.hibernate.type.DateType` 和 `org.hibernate.type.TimestampTyp` 的两个属性可能会产生类型不匹配错误，虽然数据库级可能不加区分或者可以处理这种转换。

- 对id属性来说,insert语句给你两个选择。你可以明确地在properties_list表中指定id属性（这样它的值是从对应的select表达式中获得），或者在properties_list中省略它（此时使用生成指）。后一种选择只有当使用在数据库中生成值的id产生器时才能使用；如果是“内存”中计算的类型生成器，在解析时会抛出一个异常。注意，为了说明这一问题，数据库产生值的生成器是 `org.hibernate.id.SequenceGenerator` （和它的子类），以及任何 `org.hibernate.id.PostInsertIdentifierGenerator` 接口的实现。这儿最值得注意的意外是 `org.hibernate.id.TableHiLoGenerator` ，它不能在此使用，因为它没有得到其值的途径。

- 对映射为 `version` 或 `timestamp` 的属性来说，insert语句也给你两个选择，你可以在properties_list表中指定（此时其值从对应的select表达式中获得），或者在properties_list中省略它（此时，使用在 `org.hibernate.type.VersionType` 中定义的 `seed value(种子值)` ）。

执行HQL `INSERT` 语句的例子如下：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) sel
ect c.id, c.name from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

第 14 章 HQL: Hibernate查询语言

目录

Hibernate配备了一种非常强大的查询语言，这种语言看上去很像SQL。但是不要被语法结构 上的相似所迷惑，HQL是非常有意识的被设计为完全面向对象的查询，它可以理解如继承、多态 和关联之类的概念。

14.1. 大小写敏感性问题

除了Java类与属性的名称外，查询语句对大小写并不敏感。 所以 `SeLeCT` 与 `sELEct` 以及 `SELECT` 是相同的，但是 `org.hibernate.eg.FOO` 并不等价于 `org.hibernate.eg.Foo` 并且 `foo.barSet` 也不等价于 `foo.BARSET` 。

本手册中的HQL关键字将使用小写字母. 很多用户发现使用完全大写的关键字会使查询语句 的可读性更强, 但我们发现，当把查询语句嵌入到Java语句中的时候使用大写关键字比较难看。

14.2. from子句

Hibernate中最简单的查询语句的形式如下：

```
from eg.Cat
```

该子句简单的返回 `eg.Cat` 类的所有实例。 通常我们不需要使用类的全限定名, 因为 `auto-import` （自动引入） 是缺省的情况。 所以我们几乎只使用如下的简单写法：

```
from Cat
```

大多数情况下, 你需要指定一个别名, 原因是你可能需要 在查询语句的其它部分引用到 `Cat`

```
from Cat as cat
```

这个语句把别名 `cat` 指定给类 `Cat` 的实例, 这样我们就可以在随后的查询中使用此别名了。 关键字 `as` 是可选的，我们也可以这样写:

```
from Cat cat
```

子句中可以同时出现多个类, 其查询结果是产生一个笛卡儿积或产生跨表的连接。

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

查询语句中别名的开头部分小写被认为是实践中的好习惯， 这样做与Java变量的命名标准保持了一致 (比如， `domesticCat`)。

14.3. 关联(Association)与连接(Join)

我们也可以为相关联的实体甚至是对一个集合中的全部元素指定一个别名, 这时要使用关键字 `join` 。

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

受支持的连接类型是从ANSI SQL中借鉴来的。

- `inner join` （内连接）
- `left outer join` （左外连接）
- `right outer join` （右外连接）
- `full join` (全连接，并不常用)

语句 `inner join` , `left outer join` 以及 `right outer join` 可以简写。

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

通过HQL的 `with` 关键字，你可以提供额外的join条件。

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight > 10.0
```

还有，一个"fetch"连接允许仅仅使用一个选择语句就将相关联的对象或一组值的集合随着他们的父对象的初始化而被初始化，这种方法在使用到集合的情况下尤其有用，对于关联和集合来说，它有效的代替了映射文件中的外联接 与延迟声明（lazy declarations）. 查看 第 19.1 节 “ 抓取策略(Fetching strategies) ” 以获得等多的信息。

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

一个fetch连接通常不需要被指定别名, 因为相关联的对象不应当被用在 `where` 子句 (或其它任何子句)中。同时，相关联的对象 并不在查询的结果中直接返回，但可以通过他们的父对象来访问到他们。

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

假若使用 `iterate()` 来调用查询，请注意 `fetch` 构造是不能使用的(`scroll()` 可以使用)。 `fetch` 也不应该与 `setMaxResults()` 或 `setFirstResult()` 共用，这是因为这些操作是基于结果集的，而在预先抓取集合类时可能包含重复的数据，也就是说无法预先知道精确的行数。 `fetch` 还不能与独立的 `with` 条件一起使用。通过在一次查询中fetch多个集合，可以制造出笛卡尔积，因此请多加注意。对bag映射来说，同时join fetch多个集合角色可能在某些情况下给出并非预期的结果，也请小心。最后注意，使用 `full join fetch` 与 `right join fetch` 是没有意义的。

如果你使用属性级别的延迟获取（lazy fetching）（这是通过重新编写字节码实现的），可以使用 `fetch all properties` 来强制Hibernate立即取得那些原本需要延迟加载的属性（在第一个查询中）。

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) lik
e '%cats%'
```

14.4. join 语法的形式

HQL支持两种关联join的形式：`implicit(隐式)` 与 `explicit（显式）` 。

上一节中给出的查询都是使用`explicit(显式)`形式的，其中form子句中明确给出了join关键字。这是建议使用的方式。

`implicit（隐式）` 形式不使用join关键字。关联使用"点号"来进行“引用”。 `implicit` join可以在任何HQL子句中出现. `implicit` join在最终的SQL语句中以inner join的方式出现。

```
from Cat as cat where cat.mate.name like '%s%'
```

14.5. select子句

`select` 子句选择将哪些对象与属性返 回到查询结果集中. 考虑如下情况:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

该语句将选择 `mate` s of other `Cat` s。（其他猫的配偶） 实际上, 你可以更简洁的用以下的查询语句表达相同的含义:

```
select cat.mate from Cat cat
```

查询语句可以返回值为任何类型的属性，包括返回类型为某种组件(Component)的属性:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

查询语句可以返回多个对象和（或）属性，存放在 `Object[]` 队列中,

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

或存放在一个 `List` 对象中,

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

也可能直接返回一个实际的类型安全的Java对象,

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

假设类 `Family` 有一个合适的构造函数.

你可以使用关键字 `as` 给“被选择了的表达式”指派别名:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*)
as n
from Cat cat
```

这种做法在与子句 `select new map` 一起使用时最有用:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min,
count(*) as n )
from Cat cat
```

该查询返回了一个 `Map` 的对象，内容是别名与被选择的值组成的名-值映射。

14.6. 聚集函数

HQL查询甚至可以返回作用于属性之上的聚集函数的计算结果:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(
cat)
from Cat cat
```

受支持的聚集函数如下：

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

你可以在选择子句中使用数学操作符、连接以及经过验证的SQL函数：

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

关键字 `distinct` 与 `all` 也可以使用，它们具有与SQL相同的语义.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

14.7. 多态查询

一个如下的查询语句:

```
from Cat as cat
```

不仅返回 `Cat` 类的实例, 也同时返回子类 `DomesticCat` 的实例. Hibernate 可以在 `from` 子句中指定任何 Java 类或接口. 查询会返回继承了该类的所有持久化子类 的实例或返回声明了该接口的所有持久化类的实例。下面的查询语句返回所有的被持久化的对象：

```
from java.lang.Object o
```

接口 `Named` 可能被各种各样的持久化类声明：

```
from Named n, Named m where n.name = m.name
```

注意，最后的两个查询将需要超过一个的SQL `SELECT`.这表明 `order by` 子句 没有对整个结果集进行正确的排序. (这也说明你不能对这样的查询使用 `Query.scroll()` 方法.)

14.8. where子句

`where` 子句允许你将返回的实例列表的范围缩小. 如果没有指定别名，你可以使用属性名来直接引用属性:

```
from Cat where name='Fritz'
```

如果指派了别名，需要使用完整的属性名:

```
from Cat as cat where cat.name='Fritz'
```

返回名为（属性name等于）'Fritz'的 `Cat` 类的实例。

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

将返回所有满足下面条件的 `Foo` 类的实例：存在如下的 `bar` 的一个实例，其 `date` 属性等于 `Foo` 的 `startDate` 属性。 复合路径表达式使得 `where` 子句非常的强大，考虑如下情况：

```
from Cat cat where cat.mate.name is not null
```

该查询将被翻译成为一个含有表连接（内连接）的SQL查询。如果你打算写像这样的查询语句

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

在SQL中，你为达此目的将需要进行一个四表连接的查询。

`=` 运算符不仅可以被用来比较属性的值，也可以用来比较实例：

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

特殊属性（小写）`id`可以用来表示一个对象的唯一的标识符。（你也可以使用该对象的属性名。）

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

第二个查询是有效的。此时不需要进行表连接！

同样也可以使用复合标识符。比如`Person`类有一个复合标识符，它由`country`属性与`medicareNumber`属性组成。

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

第二个查询也不需要进行表连接。

同样的，特殊属性`class`在进行多态持久化的情况下被用来存取一个实例的鉴别值（discriminator value）。一个嵌入到where子句中的Java类的名字将被转换为该类的鉴别值。

```
from Cat cat where cat.class = DomesticCat
```

你也可以声明一个属性的类型是组件或者复合用户类型（以及由组件构成的组件等等）。永远不要尝试使用以组件类型来结尾的路径表达式（path-expression）（与此相反，你应当使用组件的一个属性来结尾）。举例来说，如果`store.owner`含有一个包含了组件的实体`address`

```
store.owner.address.city    // 正确
store.owner.address         // 错误!
```

一个“任意”类型有两个特殊的属性`id`和`class`，来允许我们按照下面的方式表达一个连接（`AuditLog.item`是一个属性，该属性被映射为`<any>`）。

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

注意，在上面的查询与句中， `log.item.class` 和 `payment.class` 将涉及到完全不同的数据库中的列。

14.9. 表达式

在 `where` 子句中允许使用的表达式包括 大多数你可以在SQL使用的表达式种类:

- 数学运算符 `+, -, *, /`
- 二进制比较运算符 =, >=, <=, <>, !=, like
- 逻辑运算符 `and, or, not`
- `in` , `not in` , `between` , `is null` , `is not null` , `is empty` , `is not empty` , `member of` and `not member of`
- "简单的" case, `case ... when ... then ... else ... end` ,和 "搜索" case, `case when ... then ... else ... end`
- 字符串连接符 `...||...` or `concat(...,...)`
- `current_date()` , `current_time()` , `current_timestamp()`
- `second(...)` , `minute(...)` , `hour(...)` , `day(...)` , `month(...)` , `year(...)` ,
- EJB-QL 3.0定义的任何函数或操作： `substring(), trim(), lower(), upper(), length(), locate(), a`
- `coalesce()` 和 `nullif()`
- `str()` 把数字或者时间值转换为可读的字符串
- `cast(... as ...)` ,其第二个参数是某Hibernate类型的名字，以及 `extract(... from ...)` ，只要ANSI `cast()` 和 `extract()` 被底层数据库支持
- HQL `index()` 函数，作用于join的有序集合的别名。
- HQL函数，把集合作为参数: `size(), minelement(), maxelement(), minindex(), maxindex()` ，还有特别的 `elements()` 和 `indices` 函数，可以与数量词加以限定： `some, all, exists, any, in` 。
- 任何数据库支持的SQL标量函数，比如 `sign()` , `trunc()` , `rtrim()` , `sin()`
- JDBC风格的参数传入 `?`
- 命名参数 `:name` , `:start_date` , `:x1`
- SQL 直接常量 `'foo'` , `69` , `6.66E+2` , `'1970-01-01 10:00:01.0'`

- Java `public static final` 类型的常量 `eg.Color.TABBY`

关键字 `in` 与 `between` 可按如下方法使用:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

而且否定的格式也可以如下书写：

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz'
 )
```

同样, 子句 `is null` 与 `is not null` 可以被用来测试空值(null).

在Hibernate配置文件中声明HQL“查询替代（query substitutions）”之后， 布尔表达式（Booleans）可以在其他表达式中轻松的使用:

```
<property name="hibernate.query.substitutions">true 1, false 0</
property>
```

系统将该HQL转换为SQL语句时，该设置表明将用字符 `1` 和 `0` 来 取代关键字 `true` 和 `false`:

```
from Cat cat where cat.alive = true
```

你可以用特殊属性 `size` , 或是特殊函数 `size()` 测试一个集合的大小。

```
from Cat cat where cat.kittens.size > 0
```

```
from Cat cat where size(cat.kittens) > 0
```

对于索引了（有序）的集合，你可以使用 `minindex` 与 `maxindex` 函数来引用到最小与最大的索引序数。 同理，你可以使用 `minelement` 与 `maxelement` 函数来 引用到一个基本数据类型的集合中最小与最大的元素。

```
from Calendar cal where maxelement(cal.holidays) > current_date
```

```
from Order order where maxindex(order.items) > 100
```

```
from Order order where minelement(order.items) > 10000
```

在传递一个集合的索引集或者是元素集(`elements` 与 `indices` 函数) 或者传递一个子查询的结果的时候，可以使用SQL函数 `any, some, all, exists, in`

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3 > all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

注意，在Hibernate3种，这些结构变量- `size` , `elements` , `indices` , `minindex` , `maxindex` , `minelement` , `maxelement` - 只能在where子句中使用。

一个被索引过的（有序的）集合的元素(arrays, lists, maps)可以在其他索引中被引用（只能在where子句中）：

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and or
der.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id =
 11
```

在 `[]` 中的表达式甚至可以是一个算数表达式。

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

对于一个一对多的关联（one-to-many association）或是值的集合中的元素， HQL 也提供内建的 `index()` 函数，

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

如果底层数据库支持标量的SQL函数，它们也可以被使用

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

如果你还不能对所有的这些深信不疑，想想下面的查询。如果使用SQL，语句长度会增长多少，可读性会下降多少：

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

提示*:* 会像如下的语句

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.curren
t_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

14.10. order by子句

查询返回的列表(list)可以按照一个返回的类或组件（components)中的任何属性（property）进行排序：

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

可选的 `asc` 或 `desc` 关键字指明了按照升序或降序进行排序.

14.11. group by子句

一个返回聚集值(aggregate values)的查询可以按照一个返回的类或组件（components)中的任何属性（property）进行分组：

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

`having` 子句在这里也允许使用.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

如果底层的数据库支持的话(例如不能在MySQL中使用)，SQL的一般函数与聚集函数也可以出现 在 `having` 与 `order by` 子句中。

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

注意 `group by` 子句与 `order by` 子句中都不能包含算术表达式（arithmetic expressions）. 也要注意Hibernate目前不会扩展group的实体,因此你不能写 `group by cat` ,除非 `cat` 的所有属性都不是聚集的(non-aggregated)。你必须明确的列出所有的非聚集属性。

14.12. 子查询

对于支持子查询的数据库，Hibernate支持在查询中使用子查询。一个子查询必须被圆括号包围起来（经常是SQL聚集函数的圆括号）。 甚至相互关联的子查询（引用到外部查询中的别名的子查询）也是允许的。

```
from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

注意，HQL自查询只可以在select或者where子句中出现。

在select列表中包含一个表达式以上的子查询，你可以使用一个元组构造符（tuple constructors）：

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

注意在某些数据库中（不包括Oracle与HSQL），你也可以在其他语境中使用元组构造符， 比如查询用户类型的组件与组合：

```
from Person where name = ('Gavin', 'A', 'King')
```

该查询等价于更复杂的：

```
from Person where name.first = 'Gavin' and name.initial = 'A' an
d name.last = 'King')
```

有两个很好的理由使你不应当作这样的事情：首先，它不完全适用于各个数据库平台；其次，查询现在依赖于映射文件中属性的顺序。

14.13. HQL示例

Hibernate查询可以非常的强大与复杂。实际上，Hibernate的一个主要卖点就是查询语句的威力。这里有一些例子，它们与我在最近的 一个项目中使用的查询非常相似。注意你能用到的大多数查询比这些要简单的多！

下面的查询对于某个特定的客户的所有未支付的账单，在给定给最小总价值的情况下，返回订单的id，条目的数量和总价值， 返回值按照总价值的结果进行排序。为了决定价格，查询使用了当前目录。作为转换结果的SQL查询，使用了 `ORDER`，`ORDER_LINE`，`PRODUCT`，`CATALOG` 和 `PRICE` 库表。

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

这简直是一个怪物！实际上，在现实生活中，我并不热衷于子查询，所以我的查询语句看起来更像这个：

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

下面一个查询计算每一种状态下的支付的数目，除去所有处于 `AWAITING_APPROVAL` 状态的支付，因为在该状态下 当前的用户作出了状态的最新改变。该查询被转换成含有两个内连接以及一个相关联的子选择的SQL查询，该查询使用了表 `PAYMENT` , `PAYMENT_STATUS` 以及 `PAYMENT_STATUS_CHANGE` 。

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

如果我把 `statusChanges` 实例集映射为一个列表（list）而不是一个集合（set），书写查询语句将更加简单.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].
user <> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

下面一个查询使用了MS SQL Server的 `isNull()` 函数用以返回当前用户所属组织的组织帐号及组织未支付的账。 它被转换成一个对表 `ACCOUNT` , `PAYMENT` , `PAYMENT_STATUS` , `ACCOUNT_TYPE` , `ORGANIZATION` 以及 `ORG_USER` 进行的三个内连接， 一个外连接和一个子选择的SQL查询。

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name
, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.
dueDate
```

对于一些数据库，我们需要弃用（相关的）子选择。

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name
, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.
dueDate
```

14.14. 批量的UPDATE和DELETE

HQL现在支持 `update`, `delete` 和 `insert ... select ...`语句. 查阅 第 13.4 节 “DML(数据操作语言)风格的操作(DML-style operations)” 以获得更多信息。

14.15. 小技巧 & 小窍门

你可以统计查询结果的数目而不必实际的返回他们：

```
( (Integer) session.iterate("select count(*) from ....").next()
).intValue()
```

若想根据一个集合的大小来进行排序，可以使用如下的语句：

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

如果你的数据库支持子选择，你可以在你的查询的where子句中为选择的大小（selection size）指定一个条件:

```
from User usr where size(usr.messages) >= 1
```

如果你的数据库不支持子选择语句，使用下面的查询：

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

因为内连接（inner join）的原因，这个解决方案不能返回含有零个信息的 `User` 类的实例, 所以这种情况下使用下面的格式将是有帮助的:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

JavaBean的属性可以被绑定到一个命名查询（named query）的参数上：

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:nam
e and foo.size=:size");
q.setProperties(fooBean); // fooBean包含方法getName()与getSize()
List foos = q.list();
```

通过将接口 `Query` 与一个过滤器（filter）一起使用，集合（Collections）是可以分页的：

```
Query q = s.createFilter( collection, "" ); // 一个简单的过滤器
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

通过使用查询过滤器（query filter）可以将集合（Collection）的原素分组或排序:

```
Collection orderedCollection = s.filter( collection, "order by t
his.amount" );
Collection counts = s.filter( collection, "select this.type, cou
nt(this) group by this.type" );
```

不用通过初始化，你就可以知道一个集合（Collection）的大小：

```
( (Integer) session.iterate("select count(*) from ....").next()
).intValue();
```

第 15 章 条件查询(Criteria Queries)

目录

具有一个直观的、可扩展的条件查询API是Hibernate的特色。

15.1. 创建一个 `Criteria` 实例

`org.hibernate.Criteria`接口表示特定持久类的一个查询。`Session` 是 `Criteria` 实例的工厂。

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

15.2. 限制结果集内容

一个单独的查询条件是 `org.hibernate.criterion.Criterion` 接口的一个实例。`org.hibernate.criterion.Restrictions` 类 定义了获得某些内置 `Criterion` 类型的工厂方法。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

约束可以按逻辑分组。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi"
, "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Hibernate提供了相当多的内置criterion类型(`Restrictions` 子类), 但是尤其有用的是可以允许你直接使用SQL。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like
lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

`{alias}` 占位符应当被替换为被查询实体的列别名。

`Property` 实例是获得一个条件的另外一种途径。你可以通过调用 `Property.forName()` 创建一个 `Property` 。

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "
Izi", "Pk" } ) )
    .list();
```

15.3. 结果集排序

你可以使用 `org.hibernate.criterion.Order` 来为查询结果排序。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

15.4. 关联

你可以使用 `createCriteria()` 非常容易的在互相关联的实体间建立 约束。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

注意第二个 `createCriteria()` 返回一个新的 `Criteria` 实例，该实例引用 `kittens` 集合中的元素。

接下来，替换形态在某些情况下也是很有用的。

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` 并不创建一个新的 `Criteria` 实例。)

`Cat` 实例所保存的之前两次查询所返回的kittens集合是 没有被条件预过滤的。如果你希望只获得符合条件的kittens， 你必须使用 `ResultTransformer` 。

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

15.5. 动态关联抓取

你可以使用 `setFetchMode()` 在运行时定义动态关联抓取的语义。

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

这个查询可以通过外连接抓取 `mate` 和 `kittens`。 查看第 19.1 节 “抓取策略 (Fetching strategies)”可以获得更多信息。

15.6. 查询示例

`org.hibernate.criterion.Example` 类允许你通过一个给定实例 构建一个条件查询。

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

版本属性、标识符和关联被忽略。默认情况下值为null的属性将被排除。

你可以自行调整 `Example` 使之更实用。

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "col
or"
    .ignoreCase()              //perform case insensitive string
 comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

你甚至可以使用examples在关联对象上放置条件。

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

15.7. 投影(Projections)、聚合（aggregation）和分组（grouping）

`org.hibernate.criterion.Projections` 是 `Projection` 的实例工厂。我们通过调用 `setProjection()` 应用投影到一个查询。

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

在一个条件查询中没有必要显式的使用 "group by" 。某些投影类型就是被定义为分组投影，他们也出现在SQL的 `group by` 子句中。

你可以选择把一个别名指派给一个投影，这样可以使投影值被约束或排序所引用。下面是两种不同的实现方式：

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty
("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr"
) )
    .addOrder( Order.asc("colr") )
    .list();
```

`alias()` 和 `as()` 方法简便的将一个投影实例包装到另外一个 别名的 `Projection` 实例中。简而言之，当你添加一个投影到一个投影列表中时 你可以为它指定一个别名：

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

你也可以使用 `Property.forName()` 来表示投影：

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

15.8. 离线(detached)查询和子查询

`DetachedCriteria` 类使你在一个session范围之外创建一个查询，并且可以使用任意的 `Session` 来执行它。

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResult
s(100).list();
txn.commit();
session.close();
```

`DetachedCriteria` 也可以用以表示子查询。条件实例包含子查询可以通过 `Subqueries` 或者 `Property` 获得。

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class
)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight).gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

甚至相互关联的子查询也是有可能的：

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat
.class, "cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight).gt(avgWeightForSex) )
    .list();
```

15.9. 根据自然标识查询(Queries by natural identifier)

对大多数查询，包括条件查询而言，因为查询缓存的失效(invalidation)发生得太频繁，查询缓存不是非常高效。然而，有一种特别的查询，可以通过不变的自然键优化缓存的失效算法。在某些应用中，这种类型的查询比较常见。条件查询API对这种用例提供了特别规约。

首先，你应该对你的entity使用 `<natural-id>` 来映射自然键，然后打开第二级缓存。

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
</class>
```

注意,此功能对具有*mutable*自然键的entity并不适用。

然后，打开Hibernate 查询缓存。

现在，我们可以用 `Restrictions.naturalId()` 来使用更加高效的缓存算法。

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

第 16 章 Native SQL查询

目录

你也可以使用你的数据库的Native SQL语言来查询数据。这对你在要使用数据库的某些特性的时候(比如说在查询提示或者Oracle中的 `CONNECT` 关键字)，这是非常有用的。这就能够扫清你把原来直接使用SQL/JDBC 的程序迁移到基于 Hibernate 应用的道路上的障碍。

Hibernate3允许你使用手写的sql来完成所有的create,update,delete,和load操作（包括存储过程）

16.1. 使用 `SQLQuery`

对原生SQL查询执行的控制是通过`SQLQuery`接口进行的，通过执行`Session.createSQLQuery()`获取这个接口。下面来描述如何使用这个API进行查询。

16.1.1. 标量查询（Scalar queries）

最基本的SQL查询就是获得一个标量（数值）的列表。

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list
();
```

它们都将返回一个Object数组(Object[])组成的List，数组每个元素都是CATS表的一个字段值。Hibernate会使用ResultSetMetadata来判定返回的标量值的实际顺序和类型。

如果要避免过多的使用 `ResultSetMetadata` ,或者只是为了更加明确的指名返回值，可以使用 `addScalar()` 。

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

这个查询指定了:

- SQL查询字符串
- 要返回的字段和类型

它仍然会返回Object数组,但是此时不再使用 `ResultSetMetdata` ,而是明确的将ID,NAME和BIRTHDATE按照Long,String和Short类型从resultset中取出。同时，也指明了就算query是使用 `*` 来查询的，可能获得超过列出的这三个字段，也仅仅会返回这三个字段。

对全部或者部分的标量值不设置类型信息也是可以的。

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

基本上这和前面一个查询相同,只是此时使用 `ResultSetMetaData` 来决定NAME和BIRTHDATE的类型，而ID的类型是明确指出的。

关于从ResultSetMetaData返回的java.sql.Types是如何映射到Hibernate类型，是由方言(Dialect)控制的。假若某个指定的类型没有被映射，或者不是你所预期的类型，你可以通过Dialet的 `registerHibernateType` 调用自行定义。

16.1.2. 实体查询(Entity queries)

上面的查询都是返回标量值的，也就是从resultset中返回的“裸”数据。下面展示如何通过 `addEntity()` 让原生查询返回实体对象。

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addE
ntity(Cat.class);
```

这个查询指定：

- SQL查询字符串
- 要返回的实体

假设Cat被映射为拥有ID,NAME和BIRTHDATE三个字段的类，以上的两个查询都返回一个List，每个元素都是一个Cat实体。

假若实体在映射时有一个 `many-to-one` 的关联指向另外一个实体，在查询时必须也返回那个实体，否则会导致发生一个"column not found"的数据库错误。这些附加的字段可以使用*标注来自动返回，但我们希望还是明确指明，看下面这个具有指向 `Dog` 的 `many-to-one` 的例子：

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CAT
S").addEntity(Cat.class);
```

这样cat.getDog()就能正常运作。

16.1.3. 处理关联和集合类(Handling associations and collections)

通过提前抓取将 `Dog` 连接获得，而避免初始化proxy带来的额外开销也是可能的。这是通过 `addJoin()` 方法进行的，这个方法可以让你将关联或集合连接进来。

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID,
 D_NAME FROM CATS c, DOGS d WHERE c.DOG_ID = d.D_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

上面这个例子中，返回的 `Cat` 对象，其 `dog` 属性被完全初始化了，不再需要数据库的额外操作。注意，我们加了一个别名("cat")，以便指明join的目标属性路径。通过同样的提前连接也可以作用于集合类，例如，假若 `Cat` 有一个指向 `Dog` 的一对多关联。

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, C
AT_ID FROM CATS c, DOGS d WHERE c.ID = d.CAT_ID")
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

到此为止，我们碰到了天花板：若不对SQL查询进行增强，这些已经是在Hibernate中使用原生SQL查询所能做到的最大可能了。下面的问题即将出现：返回多个同样类型的实体怎么办？或者默认的别名/字段不够又怎么办？

16.1.4. 返回多个实体(Returning multiple entities)

到目前为止,结果集字段名被假定为和映射文件中指定的的字段名是一致的。假若SQL查询连接了多个表，同一个字段名可能在多个表中出现多次，这就会造成问题。

下面的查询中需要使用字段别名注射（这个例子本身会失败）：

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE
c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

这个查询的本意是希望每行返回两个Cat实例，一个是cat,另一个是它的妈妈。但是因为它们的字段名被映射为相同的，而且在某些数据库中，返回的字段别名是“c.ID”,"c.NAME"这样的形式，而它们和在映射文件中的名字（"ID"和"NAME"）不匹配，这就会造成失败。

下面的形式可以解决字段名重复：

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CA
TS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

这个查询指明：

- SQL查询语句，其中包含占位附来让Hibernate注射字段别名
- 查询返回的实体

上面使用的{cat.}和*{mother.}*标记是作为“所有属性”的简写形式出现的。当然你也可以明确地罗列出字段名，但在这个例子里面我们让Hibernate来为每个属性注射SQL字段别名。字段别名的占位符是属性名加上表别名的前缀。在下面的例子中，我们从另外一个表（cat_log）中通过映射元数据中的指定获取Cat和它的妈妈。注意，要是我们愿意，我们甚至可以在where子句中使用属性别名。

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {
mother.*} " +
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

16.1.4.1. 别名和属性引用(Alias and property references)

大多数情况下，都需要上面的属性注射，但在使用更加复杂的映射，比如复合属性、通过标识符构造继承树，以及集合类等等情况下，也有一些特别的别名，来允许Hibernate注射合适的别名。

下表列出了使用别名注射参数的不同可能性。注意：下面结果中的别名只是示例，实用时每个别名需要唯一并且不同的名字。

表 **16.1.** 别名注射**(alias injection names)**

描述	语法	
简单属性	`{[aliasname].[propertyname]`	A_N
复合属性	`{[aliasname].[componentname].[propertyname]}`	CUR
实体辨别器(Discriminator of an entity)	`{[aliasname].class}`	DIS
实体的所有属性	`{[aliasname].*}`	{it
集合键(collection key)	`{[aliasname].key}`	ORG
集合id	`{[aliasname].id}`	EMP
集合元素	`{[aliasname].element}`	XID
集合元素的属性	`{[aliasname].element.[propertyname]}`	NAM
集合元素的所有属性	`{[aliasname].element.*}`	{co
集合的所有属性	`{[aliasname].*}`	{co

16.1.5. 返回非受管实体(Returning non-managed entities)

可以对原生sql 查询使用ResultTransformer。这会返回不受Hibernate管理的实体。

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.cl
ass))
```

这个查询指定：

- SQL查询字符串
- 结果转换器(result transformer)

上面的查询将会返回 `CatDTO` 的列表,它将被实例化并且将NAME和BIRTHDAY的值注射入对应的属性或者字段。

16.1.6. 处理继承（Handling inheritance）

原生SQL查询假若其查询结果实体是继承树中的一部分，它必须包含基类和所有子类的所有属性。

16.1.7. 参数（Parameters）

原生查询支持位置参数和命名参数：

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME
 like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like
:name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

16.2. 命名SQL查询

可以在映射文档中定义查询的名字,然后就可以象调用一个命名的HQL查询一样直接调用命名SQL查询.在这种情况下,我们不 需要调用 addEntity() 方法.

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

<return-join> 和 <load-collection> 元素是用来连接关联以及将查询定义为预先初始化各个集合的。

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress
"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILI
NG'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

一个命名查询可能会返回一个标量值.你必须使用 <return-scalar> 元素来指定字段的别名和 Hibernate类型

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

你可以把结果集映射的信息放在外部的 `<resultset>` 元素中，这样就可以在多个命名查询间，或者通过 `setResultSetMapping()` API来访问。(此处原文即存疑。原文为：You can externalize the resultset mapping informations in a `<resultset>` element to either reuse them accross several named queries or through the `setResultSetMapping()` API.)

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress
"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILI
NG'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

另外,你可以在java代码中直接使用hbm文件中的结果集定义信息。

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten w
here kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

16.2.1. 使用return-property来明确地指定字段/别名

使用 <return-property> 你可以明确的告诉Hibernate使用哪些字段别名,这取代了使用 {} -语法 来让Hibernate注入它自己的别名.

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
      <return-property name="name" column="myName"/>
      <return-property name="age" column="myAge"/>
      <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

<return-property> 也可用于多个字段,它解决了使用 {} -语法不能细粒度控制多个字段的限制

```
<sql-query name="organizationCurrentEmployments">
            <return alias="emp" class="Employment">
             <return-property name="salary">
               <return-column name="VALUE"/>
               <return-column name="CURRENCY"/>
             </return-property>
             <return-property name="endDate" column="myEndDate"/
>
            </return>
            SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.
employer},
            STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDat
e},
            REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VAL
UE, CURRENCY
            FROM EMPLOYMENT
            WHERE EMPLOYER = :id AND ENDDATE IS NULL
            ORDER BY STARTDATE ASC
</sql-query>
```

注意在这个例子中,我们使用了 <return-property> 结合 {} 的注入语法. 允许用户来选择如何引用字段以及属性.

如果你映射一个识别器(discriminator),你必须使用 `<return-discriminator>` 来指定识别器字段

16.2.2. 使用存储过程来查询

Hibernate 3引入了对存储过程查询(stored procedure)和函数(function)的支持.以下的说明中，这二者一般都适用。 存储过程/函数必须返回一个结果集,作为Hibernate能够使用的第一个外部参数. 下面是一个Oracle9和更高版本的存储过程例子.

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

在Hibernate里要要使用这个查询,你需要通过命名查询来映射它.

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>

        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>

        <return-property name="regionCode" column="REGIONCODE"/>

        <return-property name="id" column="EID"/>

        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

注意存储过程当前仅仅返回标量和实体.现在不支持 <return-join> 和 <load-collection>

16.2.2.1. 使用存储过程的规则和限制

为了在Hibernate中使用存储过程,你必须遵循一些规则.不遵循这些规则的存储过程将不可用.如果你仍然想要使用他们, 你必须通过 `session.connection()` 来执行他们.这些规则针对于不同的数据库.因为数据库 提供商有各种不同的存储过程语法和语义.

对存储过程进行的查询无法使用 `setFirstResult()/setMaxResults()` 进行分页。

建议采用的调用方式是标准SQL92: `{ ? = call functionName(<parameters>) }` 或者 `{ ? = call procedureName(<parameters>}` .原生调用语法不被支持。

对于Oracle有如下规则:

- 函数必须返回一个结果集。存储过程的第一个参数必须是 `OUT` ，它返回一个结果集。这是通过Oracle 9或10的 `SYS_REFCURSOR` 类型来完成的。在Oracle中你需要定义一个 `REF CURSOR` 类型，参见Oracle的手册。

对于Sybase或者MS SQL server有如下规则:

- 存储过程必须返回一个结果集。.注意这些servers可能返回多个结果集以及更新的数目.Hibernate将取出第一条结果集作为它的返回值， 其他将被丢弃。
- 如果你能够在存储过程里设定 `SET NOCOUNT ON` ，这可能会效率更高，但这不是必需的。

16.3. 定制SQL用来create，update和delete

Hibernate3能够使用定制的SQL语句来执行create,update和delete操作。在Hibernate中，持久化的类和集合已经 包含了一套配置期产生的语句(insertsql, deletesql, updatesql等等)，这些映射标记 <sql-insert> , <sql-delete> , and <sql-update> 重载了 这些语句。

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?),
 ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-
update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

这些SQL直接在你的数据库里执行，所以你可以自由的使用你喜欢的任意语法。但如果你使用数据库特定的语法， 这当然会降低你映射的可移植性。

如果设定 `callable` ，则能够支持存储过程了。

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert callable="true">{call createPerson (?, ?)}</sql-
insert>
    <sql-delete callable="true">{? = call deletePerson (?)}</sql
-delete>
    <sql-update callable="true">{? = call updatePerson (?, ?)}</
sql-update>
</class>
```

参数的位置顺序是非常重要的，他们必须和Hibernate所期待的顺序相同。

你能够通过设定日志调试级别为 `org.hiberante.persister.entity` ,来查看Hibernate所期待的顺序。在这个级别下， Hibernate将会打印出create,update和delete实体的静态SQL。(如果想看到预计的顺序。记得不要将定制SQL包含在映射文件里， 因为他们会重载Hibernate生成的静态SQL。)

在大多数情况下(最好这么做)，存储过程需要返回插入/更新/删除的行数，因为Hibernate对语句的成功执行有些运行时的检查。 Hibernate常会把进行CUD操作的语句的第一个参数注册为一个数值型输出参数。

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN
 VARCHAR2)
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

16.4. 定制装载SQL

你可能需要声明你自己的SQL(或HQL)来装载实体

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

这只是一个前面讨论过的命名查询声明，你可以在类映射里引用这个命名查询。

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

这也可以用于存储过程

你甚至可以定一个用于集合装载的查询:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

你甚至还可以定义一个实体装载器，它通过连接抓取装载一个集合:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

第 17 章 过滤数据

目录

Hibernate3 提供了一种创新的方式来处理具有“显性(visibility)”规则的数据，那就是使用*Hibernate filter*。 *Hibernate filter*是全局有效的、具有名字、可以带参数的过滤器， 对于某个特定的Hibernate session您可以选择是否启用（或禁用）某个过滤器。

17.1. Hibernate 过滤器(filters)

Hibernate3新增了对某个类或者集合使用预先定义的过滤器条件(filter criteria)的功能。过滤器条件相当于定义一个 非常类似于类和各种集合上的“where”属性的约束子句，但是过滤器条件可以带参数。 应用程序可以在运行时决定是否启用给定的过滤器，以及使用什么样的参数值。 过滤器的用法很像数据库视图，只不过是在应用程序中确定使用什么样的参数的。

要使用过滤器，必须首先在相应的映射节点中定义。而定义一个过滤器，要用到位于 `<hibernate-mapping/>` 节点之内的 `<filter-def/>` 节点：

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

定义好之后，就可以在某个类中使用这个过滤器：

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTE
RED_COLUMN"/>
</class>
```

也可以在某个集合使用它：

```
<set ...>
    <filter name="myFilter" condition=":myFilterParam = MY_FILTE
RED_COLUMN"/>
</set>
```

可以在多个类或集合中使用某个过滤器；某个类或者集合中也可以使用多个过滤器。

`Session` 对象中会用到的方法有： `enableFilter(String filterName)` , `getEnabledFilter(String filterName)` , 和 `disableFilter(String filterName)` . Session中默认是不启用过滤器的，必须通过 `Session.enabledFilter()` 方法显式的启用。 该方法返回被启用的 `Filter` 的实例。以上文定义的过滤器为例：

```
session.enableFilter("myFilter").setParameter("myFilterParam", "
some-value");
```

注意，org.hibernate.Filter的方法允许链式方法调用。（类似上面例子中启用Filter之后设定Filter参数这个“方法链”） Hibernate的其他部分也大多有这个特性。

下面是一个比较完整的例子，使用了记录生效日期模式过滤有时效的数据：

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
    <many-to-one name="department" column="dept_id" class="Depar
tment"/>
    <property name="effectiveStartDate" type="date" column="eff_
start_dt"/>
    <property name="effectiveEndDate" type="date" column="eff_en
d_dt"/>
...
    <!--
        Note that this assumes non-terminal records have an eff_
end_dt set to
        a max db date for simplicity-sake

        注意，为了简单起见，此处假设雇用关系生效期尚未结束的记录的eff_end_
dt字段的值等于数据库最大的日期
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_en
d_dt"/>
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and ef
f_end_dt"/>
    </set>
</class>
```

定义好后，如果想要保证取回的都是目前处于生效期的记录，只需在获取雇员数据的操作之前先开启过滤器即可：

```
Session session = ...;
session.enabledFilter("effectiveDate").setParameter("asOfDate",
new Date());
List results = session.createQuery("from Employee as e where e.s
alary > :targetSalary")
         .setLong("targetSalary", new Long(1000000))
         .list();
```

在上面的HQL中，虽然我们仅仅显式的使用了一个薪水条件，但因为启用了过滤器，查询将仅返回那些目前雇用 关系处于生效期的，并且薪水高于一百万美刀的雇员的数据。

注意：如果你打算在使用外连接（或者通过HQL或load fetching）的同时使用过滤器，要注意条件表达式的方向（左还是右）。 最安全的方式是使用左外连接（left outer joining）。并且通常来说，先写参数， 然后是操作符，最后写数据库字段名。

在Filter定义之后,它可能被附加到多个实体和/或集合类,每个都有自己的条件。假若这些条件都是一样的，每次都要定义就显得很繁琐。因此， `<filter-def/>` 被用来定义一个默认条件，它可能作为属性或者CDATA出现：

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-de
f>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

当这个filter被附加到任何目的地，而又没有指明条件时，这个条件就会被使用。注意，换句话说，你可以通过给filter附加特别的条件来重载默认条件。

第 18 章 XML映射

目录

注意这是*Hibernate 3.0*的一个实验性的特性。这一特性仍在积极开发中。

18.1. 用XML数据进行工作

Hibernate使得你可以用XML数据来进行工作，恰如你用持久化的POJO进行工作那样。解析过的XML树 可以被认为是代替POJO的另外一种在对象层面上表示关系型数据的途径.

Hibernate支持采用dom4j作为操作XML树的API。你可以写一些查询从数据库中检索出 dom4j树，随后你对这颗树做的任何修改都将自动同步回数据库。你甚至可以用dom4j解析 一篇XML文档，然后使用Hibernate的任一基本操作将它写入数据库：`persist(), saveOrUpdate(), merge(), delete(), replicate()` (合并操作merge()目前还不支持)。

这一特性可以应用在很多场合，包括数据导入导出，通过JMS或SOAP具体化实体数据以及 基于XSLT的报表。

一个单一的映射就可以将类的属性和XML文档的节点同时映射到数据库。如果不需要映射类， 它也可以用来只映射XML文档。

18.1.1. 指定同时映射XML和类

这是一个同时映射POJO和XML的例子：

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...

</class>
```

18.1.2. 只定义XML映射

这是一个不映射POJO的例子：

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class>
```

这个映射使得你既可以把数据作为一棵dom4j树那样访问，又可以作为由属性键值对(java `Map` s) 组成的图那样访问。属性名字纯粹是逻辑上的结构，你可以在HQL查询中引用它。

18.2. XML映射元数据

许多Hibernate映射元素具有 `node` 属性。这使你可以指定用来保存 属性或实体数据的XML属性或元素。 `node` 属性必须是下列格式之一：

- `"element-name"` - 映射为指定的XML元素
- `"@attribute-name"` - 映射为指定的XML属性
- `"."` - 映射为父元素
- `"element-name/@attribute-name"` - 映射为指定元素的指定属性

对于集合和单值的关联，有一个额外的 `embed-xml` 属性可用。 这个属性的缺省值是真(`embed-xml="true"`)。如果 `embed-xml="true"` ， 则对应于被关联实体或值类型的集合的XML树将直接嵌入拥有这些关联的实体的XML树中。 否则，如果 `embed-xml="false"` ，那么对于单值的关联，仅被引用的实体的标识符出现在XML树中(被引用实体本身不出现)，而集合则根本不出现。

你应该小心，不要让太多关联的embed-xml属性为真(`embed-xml="true"`)，因为XML不能很好地处理 循环引用!

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>

    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
                type="string"/>
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class>
```

在这个例子中，我们决定嵌入帐目号码(account id)的集合，但不嵌入实际的帐目数据。下面的HQL查询：

```
from Customer c left join fetch c.accounts where c.lastName like
 :lastName
```

返回的数据集将是这样：

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings"/>
    <account id="985612323" short-desc="Credit Card"/>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

如果你把一对多映射 <one-to-many> 的embed-xml属性置为真(`embed-xml="true"`)，则数据看上去就像这样：

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance>-2370.34</balance>
    </account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

18.3. 操作XML数据

让我们来读入和更新应用程序中的XML文档。通过获取一个dom4j会话可以做到这一点：

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts whe
re c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId
);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

将这一特色与Hibernate的 `replicate()` 操作结合起来对于实现的基于XML的数据导入/导出将非常有用.

第 19 章 提升性能

目录

19.1. 抓取策略(Fetching strategies)

抓取策略（*fetching strategy*） 是指：当应用程序需要在（Hibernate实体对象图的）关联关系间进行导航的时候， Hibernate如何获取关联对象的策略。抓取策略可以在O/R映射的元数据中声明，也可以在特定的HQL或`条件查询（Criteria Query）`中重载声明。

Hibernate3 定义了如下几种抓取策略：

- 连接抓取（*Join fetching*） - Hibernate通过 在`SELECT`语句使用`OUTER JOIN`（外连接）来 获得对象的关联实例或者关联集合。
- 查询抓取（*Select fetching*） - 另外发送一条 `SELECT` 语句抓取当前对象的关联实体或集合。除非你显式的指定`lazy="false"`禁止 延迟抓取（lazy fetching），否则只有当你真正访问关联关系的时候，才会执行第二条select语句。
- 子查询抓取（*Subselect fetching*） - 另外发送一条`SELECT` 语句抓取在前面查询到（或者抓取到）的所有实体对象的关联集合。除非你显式的指定`lazy="false"` 禁止延迟抓取（lazy fetching），否则只有当你真正访问关联关系的时候，才会执行第二条select语句。
- 批量抓取（*Batch fetching*） - 对查询抓取的优化方案， 通过指定一个主键或外键列表，Hibernate使用单条`SELECT`语句获取一批对象实例或集合。

Hibernate会区分下列各种情况：

- *Immediate fetching*，立即抓取 - 当宿主被加载时，关联、集合或属性被立即抓取。
- *Lazy collection fetching*，延迟集合抓取- 直到应用程序对集合进行了一次操作时，集合才被抓取。（对集合而言这是默认行为。）
- *"Extra-lazy" collection fetching,"Extra-lazy"*集合抓取 -对集合类中的每个元素而言，都是直到需要时才去访问数据库。除非绝对必要，Hibernate不会试图去把整个集合都抓取到内存里来（适用于非常大的集合）。
- *Proxy fetching*，代理抓取 - 对返回单值的关联而言，当其某个方法被调用，而非对其关键字进行get操作时才抓取。
- *"No-proxy" fetching,*非代理抓取 - 对返回单值的关联而言，当实例变量被访问的时候进行抓取。与上面的代理抓取相比，这种方法没有那么“延迟”得厉害(就算只访问标识符，也会导致关联抓取)但是更加透明，因为对应用程序来说，不再看到proxy。这种方法需要在编译期间进行字节码增强操作，因此很少需要用到。
- *Lazy attribute fetching*，属性延迟加载 - 对属性或返回单值的关联而言，当其实例变量被访问的时候进行抓取。需要编译期字节码强化，因此这一方法很少是必要的。

这里有两个正交的概念：关联何时被抓取，以及被如何抓取（会采用什么样的SQL语句）。不要混淆它们！我们使用 `抓取` 来改善性能。我们使用 `延迟` 来定义一些契约，对某特定类的某个脱管的实例，知道有哪些数据是可以使用的。

19.1.1. 操作延迟加载的关联

默认情况下，Hibernate 3对集合使用延迟select抓取，对返回单值的关联使用延迟代理抓取。对几乎是所有的应用而言，其绝大多数的关联，这种策略都是有效的。

注意*:*假若你设置了 `hibernate.default_batch_fetch_size` ,Hibernate会对延迟加载采取批量抓取优化措施（这种优化也可能会在更细化的级别打开）。

然而，你必须了解延迟抓取带来的一个问题。在一个打开的Hibernate session上下文之外调用延迟集合会导致一次意外。比如：

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userNam
e")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  //
 Error!
```

在 `Session` 关闭后，permessions集合将是未实例化的、不再可用，因此无法正常载入其状态。 *Hibernate*对脱管对象不支持延迟实例化. 这里的修改方法是：将permissions读取数据的代码 移到tx.commit()之前。

除此之外，通过对关联映射指定 `lazy="false"` ,我们也可以使用非延迟的集合或关联。但是， 对绝大部分集合来说，更推荐使用延迟方式抓取数据。如果在你的对象模型中定义了太多的非延迟关联，Hibernate最终几乎需要在每个事务中载入整个数据库到内存中！

但是，另一方面，在一些特殊的事务中，我们也经常需要使用到连接抓取（它本身上就是非延迟的），以代替查询抓取。 下面我们将会很快明白如何具体的定制Hibernate中的抓取策略。在Hibernate3中，具体选择哪种抓取策略的机制是和选择单值关联或集合关联相一致的。

19.1.2. 调整抓取策略（Tuning fetch strategies）

查询抓取（默认的）在N+1查询的情况下是极其脆弱的，因此我们可能会要求在映射文档中定义使用连接抓取：

```
<set name="permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

在映射文档中定义的 `抓取` 策略将会对以下列表条目产生影响：

- 通过 `get()` 或 `load()` 方法取得数据。
- 只有在关联之间进行导航时，才会隐式的取得数据。
- `条件查询`
- 使用了 `subselect` 抓取的HQL查询

不管你使用哪种抓取策略，定义为非延迟的类图会被保证一定装载入内存。注意这可能意味着在一条HQL查询后紧跟着一系列的查询。

通常情况下，我们并不使用映射文档进行抓取策略的定制。更多的是，保持其默认值，然后在特定的事务中， 使用HQL的 `左连接抓取（left join fetch）` 对其进行重载。这将通知 Hibernate在第一次查询中使用外部关联（outer join），直接得到其关联数据。 在 `条件查询` API中，应该调用 `setFetchMode(FetchMode.JOIN)` 语句。

也许你喜欢仅仅通过条件查询，就可以改变 `get()` 或 `load()` 语句中的数据抓取策略。例如：

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
                .uniqueResult();
```

（这就是其他ORM解决方案的“抓取计划(fetch plan)”在Hibernate中的等价物。）

截然不同的一种避免N+1次查询的方法是，使用二级缓存。

19.1.3. 单端关联代理（Single-ended association proxies）

在Hinerbate中，对集合的延迟抓取的采用了自己的实现方法。但是，对于单端关联的延迟抓取，则需要采用 其他不同的机制。单端关联的目标实体必须使用代理，Hihernate在运行期二进制级（通过优异的CGLIB库）， 为持久对象实现了延迟载入代理。

默认的，Hibernate3将会为所有的持久对象产生代理（在启动阶段），然后使用他们实现 `多对一（many-to-one）` 关联和 `一对一（one-to-one）` 关联的延迟抓取。

在映射文件中，可以通过设置 `proxy` 属性为目标class声明一个接口供代理接口使用。 默认的，Hibernate将会使用该类的一个子类。 注意：被代理的类必须实现一个至少包可见的默认构造函数，我们建议所有的持久类都应拥有这样的构造函数

在如此方式定义一个多态类的时候，有许多值得注意的常见性的问题，例如：

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
```

首先， `Cat` 实例永远不可以被强制转换为 `DomesticCat` ，即使它本身就是 `DomesticCat` 实例。

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a p
roxy (does not hit the db)
if ( cat.isDomesticCat() ) {                  // hit the db to i
nitialize the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

其次，代理的“ `==` ”可能不再成立。

```
Cat cat = (Cat) session.load(Cat.class, id);            // insta
ntiate a Cat proxy
DomesticCat dc =
        (DomesticCat) session.load(DomesticCat.class, id);  // a
cquire new DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

虽然如此，但实际情况并没有看上去那么糟糕。虽然我们现在有两个不同的引用，分别指向这两个不同的代理对象， 但实际上，其底层应该是同一个实例对象：

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

第三，你不能对“final类”或“具有final方法的类”使用CGLIB代理。

最后，如果你的持久化对象在实例化时需要某些资源（例如，在实例化方法、默认构造方法中）， 那么代理对象也同样需要使用这些资源。实际上，代理类是持久化类的子类。

这些问题都源于Java的单根继承模型的天生限制。如果你希望避免这些问题，那么你的每个持久化类必须实现一个接口， 在此接口中已经声明了其业务方法。然后，你需要在映射文档中再指定这些接口。例如：

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

这里 `CatImpl` 实现了 `Cat` 接口， `DomesticCatImpl` 实现 `DomesticCat` 接口。 在 `load()` 、 `iterate()` 方法中就会返回 `Cat` 和 `DomesticCat` 的代理对象。(注意 `list()` 并不会返回代理对象。)

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.iterate("from CatImpl as cat where cat.n
ame='fritz'");
Cat fritz = (Cat) iter.next();
```

这里，对象之间的关系也将被延迟载入。这就意味着，你应该将属性声明为 `Cat` ，而不是 `CatImpl` 。

但是，在有些方法中是不需要使用代理的。例如：

- `equals()` 方法，如果持久类没有重载 `equals()` 方法。
- `hashCode()` 方法，如果持久类没有重载 `hashCode()` 方法。
- 标志符的getter方法。

Hibernate将会识别出那些重载了 `equals()` 、或 `hashCode()` 方法的持久化类。

若选择 `lazy="no-proxy"` 而非默认的 `lazy="proxy"` ，我们可以避免类型转换带来的问题。然而，这样我们就需要编译期字节码增强，并且所有的操作都会导致立刻进行代理初始化。

19.1.4. 实例化集合和代理（Initializing collections and proxies）

在 `Session` 范围之外访问未初始化的集合或代理，Hibernate将会抛出 `LazyInitializationException` 异常。 也就是说，在分离状态下，访问一个实体所拥有的集合，或者访问其指向代理的属性时，会引发此异常。

有时候我们需要保证某个代理或者集合在Session关闭前就已经被初始化了。 当然，我们可以通过强行调用 `cat.getSex()` 或者 `cat.getKittens().size()` 之类的方法来确保这一点。 但是这样的程序会造成读者的疑惑，也不符合通常的代码规范。

静态方法 `Hibernate.initialized()` 为你的应用程序提供了一个便捷的途径来延迟加载集合或代理。 只要它的Session处于open状态， `Hibernate.initialize(cat)` 将会为cat强制对代理实例化。 同样， `Hibernate.initialize( cat.getKittens() )` 对kittens的集合具有同样的功能。

还有另外一种选择，就是保持 `Session` 一直处于open状态，直到所有需要的集合或代理都被载入。 在某些应用架构中，特别是对于那些使用Hibernate进行数据访问的代码，以及那些在不同应用层和不同物理进程中使用Hibernate的代码。 在集合实例化时，如何保证 `Session` 处于open状态经常会是一个问题。有两种方法可以解决此问题：

- 在一个基于Web的应用中，可以利用servlet过滤器（filter），在用户请求（request）结束、页面生成 结束时关闭 `Session` （这里使用了在展示层保持打开*Session*模式（*Open Session in View*）），当然，这将依赖于应用框架中异常需要被正确的处理。在返回界面给用户之前，乃至在生成界面过程中发生异常的情况下， 正确关闭 `Session` 和结束事务将是非常重要的， 请参见Hibernate wiki上的"Open Session in View"模式，你可以找到示例。
- 在一个拥有单独业务层的应用中，业务层必须在返回之前，为web层“准备”好其所需的数据集合。这就意味着 业务层应该载入所有表现层/web层所需的数据，并将这些已实例化完毕的数据返回。通常，应用程序应该 为web层所需的每个集合调用 `Hibernate.initialize()` （这个调用必须发生咱session关闭之前）； 或者使用带有 `FETCH` 从句，或 `FetchMode.JOIN` 的Hibernate查询， 事先取得所有的数据集合。如果你在应用中使用了*Command*模式，代替*Session Facade* ， 那么这项任务将会变得简单的多。
- 你也可以通过 `merge()` 或 `lock()` 方法，在访问未实例化的集合（或代理）之前， 为先前载入的对象绑定一个新的 `Session` 。 显然，Hibernate将不会，也不应该自动完成这些任务，因为这将引入一个特殊的事务语义。

有时候，你并不需要完全实例化整个大的集合，仅需要了解它的部分信息（例如其大小）、或者集合的部分内容。

你可以使用集合过滤器得到其集合的大小，而不必实例化整个集合：

```
( (Integer) s.createFilter( collection, "select count(*)" ).list
().get(0) ).intValue()
```

这里的 `createFilter()` 方法也可以被用来有效的抓取集合的部分内容，而无需实例化整个集合：

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResu
lts(10).list();
```

19.1.5. 使用批量抓取（Using batch fetching）

Hibernate可以充分有效的使用批量抓取，也就是说，如果仅一个访问代理（或集合），那么Hibernate将不载入其他未实例化的代理。 批量抓取是延迟查询抓取的优化方案，你可以在两种批量抓取方案之间进行选择：在类级别和集合级别。

类/实体级别的批量抓取很容易理解。假设你在运行时将需要面对下面的问题：你在一个 `Session` 中载入了25个 `Cat` 实例，每个 `Cat` 实例都拥有一个引用成员 `owner` ，其指向 `Person` ，而 `Person` 类是代理，同时 `lazy="true"` 。 如果你必须遍历整个cats集合，对每个元素调用 `getOwner()` 方法，Hibernate将会默认的执行25次 `SELECT` 查询， 得到其owner的代理对象。这时，你可以通过在映射文件的 `Person` 属性，显式声明 `batch-size` ，改变其行为：

```
<class name="Person" batch-size="10">...</class>
```

随之，Hibernate将只需要执行三次查询，分别为10、10、 5。

你也可以在集合级别定义批量抓取。例如，如果每个 `Person` 都拥有一个延迟载入的 `Cats` 集合， 现在， `Sesssion` 中载入了10个person对象，遍历person集合将会引起10次 `SELECT` 查询， 每次查询都会调用 `getCats()` 方法。如果你在 `Person` 的映射定义部分，允许对 `cats` 批量抓取, 那么，Hibernate将可以预先抓取整个集合。请看例子：

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

如果整个的 `batch-size` 是3（笔误？），那么Hibernate将会分四次执行 `SELECT` 查询， 按照3、3、3、1的大小分别载入数据。这里的每次载入的数据量还具体依赖于当前 `Session` 中未实例化集合的个数。

如果你的模型中有嵌套的树状结构，例如典型的帐单－原料结构（bill-of-materials pattern），集合的批量抓取是非常有用的。（尽管在更多情况下对树进行读取时，嵌套集合（*nested set*）或原料路径*(materialized path)*（××） 是更好的解决方法。）

19.1.6. 使用子查询抓取（Using subselect fetching）

假若一个延迟集合或单值代理需要抓取，Hibernate会使用一个subselect重新运行原来的查询，一次性读入所有的实例。这和批量抓取的实现方法是一样的，不会有破碎的加载。

19.1.7. 使用延迟属性抓取（Using lazy property fetching）

Hibernate3对单独的属性支持延迟抓取，这项优化技术也被称为组抓取（*fetch groups*）。 请注意，该技术更多的属于市场特性。在实际应用中，优化行读取比优化列读取更重要。但是，仅载入类的部分属性在某些特定情况下会有用，例如在原有表中拥有几百列数据、数据模型无法改动的情况下。

可以在映射文件中对特定的属性设置 `lazy` ，定义该属性为延迟载入。

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="
true"/>
    <property name="text" not-null="true" length="2000" lazy="tr
ue"/>
</class>
```

属性的延迟载入要求在其代码构建时加入二进制指示指令（bytecode instrumentation），如果你的持久类代码中未含有这些指令， Hibernate将会忽略这些属性的延迟设置，仍然将其直接载入。

你可以在Ant的Task中，进行如下定义，对持久类代码加入“二进制指令。”

```
<target name="instrument" depends="compile">
    <taskdef name="instrument" classname="org.hibernate.tool.ins
trument.InstrumentTask">
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/m
odel">
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

还有一种可以优化的方法，它使用HQL或条件查询的投影（projection）特性，可以避免读取非必要的列， 这一点至少对只读事务是非常有用的。它无需在代码构建时“二进制指令”处理，因此是一个更加值得选择的解决方法。

有时你需要在HQL中通过`抓取所有属性`，强行抓取所有内容。

19.2. 二级缓存（The Second Level Cache）

Hibernate的 `Session` 在事务级别进行持久化数据的缓存操作。 当然，也有可能分别为每个类（或集合)，配置集群、或JVM级别(`SessionFactory级别`)的缓存。你甚至可以为之插入一个集群的缓存。注意，缓存永远不知道其他应用程序对持久化仓库（数据库）可能进行的修改 （即使可以将缓存数据设定为定期失效）。

通过在 `hibernate.cache.provider_class` 属性中指定 `org.hibernate.cache.CacheProvider` 的某个实现的类名,你可以选择让Hibernate使用哪个缓存实现。Hibernate打包一些开源缓存实现，提供对它们的内置支持（见下表）。除此之外，你也可以实现你自己的实现，将它们插入到系统中。注意，在3.2版本之前，默认使用EhCache 作为缓存实现，但从3.2起就不再这样了。

表 **19.1.** 缓存策略提供商（**Cache Providers**）

Cache	Provider class	Typ
Hashtable (not intended for production use)	`org.hibernate.cache.HashtableCacheProvider`	memor
EHCache	`org.hibernate.cache.EhCacheProvider`	memor disk
OSCache	`org.hibernate.cache.OSCacheProvider`	memor disk
SwarmCache	`org.hibernate.cache.SwarmCacheProvider`	clustere multica
JBoss TreeCache	`org.hibernate.cache.TreeCacheProvider`	clustere multica transac

19.2.1. 缓存映射（Cache mappings）

类或者集合映射的“ <cache> 元素”可以有下列形式：

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-on
ly"
    region="RegionName"
    include="all|non-lazy"
/>
```

❶	**usage** **(**必须**)**说明了缓存的策略**:** **transactional** 、 **read-write** 、 **nonstrict-read-write** 或 **read-only** 。
❷	region (可选, 默认为类或者集合的名字(class or collection role name)) 指定第二级缓存的区域名(name of the second level cache region)
❸	include (可选,默认为 all) non-lazy 当属性级延迟抓取打开时, 标记为 lazy="true" 的实体的属性可能无法被缓存

另外(首选?), 你可以在hibernate.cfg.xml中指定 <class-cache> 和 <collection-cache> 元素。

这里的 usage 属性指明了缓存并发策略（*cache concurrency strategy*）。

19.2.2. 策略：只读缓存（Strategy: read only）

如果你的应用程序只需读取一个持久化类的实例，而无需对其修改， 那么就可以对其进行 `只读` 缓存。这是最简单，也是实用性最好的方法。甚至在集群中，它也能完美地运作。

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

19.2.3. 策略:读/写缓存（Strategy: read/write）

如果应用程序需要更新数据，那么使用 `读/写缓存` 比较合适。 如果应用程序要求“序列化事务”的隔离级别（serializable transaction isolation level），那么就决不能使用这种缓存策略。 如果在JTA环境中使用缓存，你必须指定 `hibernate.transaction.manager_lookup_class` 属性的值， 通过它，Hibernate才能知道该应用程序中JTA的 `TransactionManager` 的具体策略。 在其它环境中，你必须保证在 `Session.close()` 、或 `Session.disconnect()` 调用前， 整个事务已经结束。 如果你想在集群环境中使用此策略，你必须保证底层的缓存实现支持锁定(locking)。Hibernate内置的缓存策略并不支持锁定功能。

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

19.2.4. 策略:非严格读/写缓存（Strategy: nonstrict read/write）

如果应用程序只偶尔需要更新数据（也就是说，两个事务同时更新同一记录的情况很不常见），也不需要十分严格的事务隔离， 那么比较适合使用 `非严格读/写缓存` 策略。如果在JTA环境中使用该策略， 你必须为其指定 `hibernate.transaction.manager_lookup_class` 属性的值， 在其它环境中，你必须保证在 `Session.close()` 、或 `Session.disconnect()` 调用前， 整个事务已经结束。

19.2.5. 策略:事务缓存（transactional）

Hibernate的 `事务缓存` 策略提供了全事务的缓存支持， 例如对JBoss TreeCache的支持。这样的缓存只能用于JTA环境中，你必须指定 为其 `hibernate.transaction.manager_lookup_class` 属性。

没有一种缓存提供商能够支持上列的所有缓存并发策略。下表中列出了各种提供器、及其各自适用的并发策略。

表 19.2. 各种缓存提供商对缓存并发策略的支持情况（Cache Concurrency Strategy Support）

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes	yes		

19.3. 管理缓存（Managing the caches）

无论何时，当你给 `save()` 、 `update()` 或 `saveOrUpdate()` 方法传递一个对象时，或使用 `load()` 、 `get()` 、 `list()` 、 `iterate()` 或 `scroll()` 方法获得一个对象时, 该对象都将被加入到 `Session` 的内部缓存中。

当随后flush()方法被调用时，对象的状态会和数据库取得同步。 如果你不希望此同步操作发生，或者你正处理大量对象、需要对有效管理内存时，你可以调用 `evict()` 方法，从一级缓存中去掉这些对象及其集合。

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scro
ll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

Session还提供了一个 `contains()` 方法，用来判断某个实例是否处于当前session的缓存中。

如若要把所有的对象从session缓存中彻底清除，则需要调用 `Session.clear()` 。

对于二级缓存来说，在 `SessionFactory` 中定义了许多方法， 清除缓存中实例、整个类、集合实例或者整个集合。

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a 
particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitte
n collections
```

`CacheMode` 参数用于控制具体的Session如何与二级缓存进行交互。

- `CacheMode.NORMAL` - 从二级缓存中读、写数据。
- `CacheMode.GET` - 从二级缓存中读取数据，仅在数据更新时对二级缓存写数据。
- `CacheMode.PUT` - 仅向二级缓存写数据，但不从二级缓存中读数据。
- `CacheMode.REFRESH` - 仅向二级缓存写数据，但不从二级缓存中读数据。通过 `hibernate.cache.use_minimal_puts` 的设置，强制二级缓存从数据库中读取数据，刷新缓存内容。

如若需要查看二级缓存或查询缓存区域的内容，你可以使用 `统计（Statistics）` API。

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

此时，你必须手工打开统计选项。可选的，你可以让Hibernate更人工可读的方式维护缓存内容。

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

19.4. 查询缓存（The Query Cache）

查询的结果集也可以被缓存。只有当经常使用同样的参数进行查询时，这才会有些用处。 要使用查询缓存，首先你必须打开它：

```
hibernate.cache.use_query_cache true
```

该设置将会创建两个缓存区域 - 一个用于保存查询结果集(`org.hibernate.cache.StandardQueryCache`)； 另一个则用于保存最近查询的一系列表的时间戳(`org.hibernate.cache.UpdateTimestampsCache`)。 请注意：在查询缓存中，它并不缓存结果集中所包含的实体的确切状态；它只缓存这些实体的标识符属性的值、以及各值类型的结果。 所以查询缓存通常会和二级缓存一起使用。

绝大多数的查询并不能从查询缓存中受益，所以Hibernate默认是不进行查询缓存的。如若需要进行缓存，请调用 `Query.setCacheable(true)` 方法。这个调用会让查询在执行过程中时先从缓存中查找结果， 并将自己的结果集放到缓存中去。

如果你要对查询缓存的失效政策进行精确的控制，你必须调用 `Query.setCacheRegion()` 方法， 为每个查询指定其命名的缓存区域。

```
List blogs = sess.createQuery("from Blog blog where blog.blogger
 = :blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

如果查询需要强行刷新其查询缓存区域，那么你应该调用 `Query.setCacheMode(CacheMode.REFRESH)` 方法。 这对在其他进程中修改底层数据（例如，不通过Hibernate修改数据），或对那些需要选择性更新特定查询结果集的情况特别有用。 这是对 `SessionFactory.evictQueries()` 的更为有效的替代方案，同样可以清除查询缓存区域。

19.5. 理解集合性能（Understanding Collection performance）

前面我们已经对集合进行了足够的讨论。本段中，我们将着重讲述集合在运行时的事宜。

19.5.1. 分类（Taxonomy）

Hibernate定义了三种基本类型的集合：

- 值数据集合
- 一对多关联
- 多对多关联

这个分类是区分了不同的表和外键关系类型，但是它没有告诉我们关系模型的所有内容。 要完全理解他们的关系结构和性能特点，我们必须同时考虑“用于Hibernate更新或删除集合行数据的主键的结构”。 因此得到了如下的分类：

- 有序集合类
- 集合（sets）
- 包（bags)

所有的有序集合类（maps, lists, arrays)都拥有一个由 `<key>` 和 `<index>` 组成的主键。 这种情况下集合类的更新是非常高效的——主键已经被有效的索引，因此当Hibernate试图更新或删除一行时，可以迅速找到该行数据。

集合(sets)的主键由 `<key>` 和其他元素字段构成。 对于有些元素类型来说，这很低效，特别是组合元素或者大文本、大二进制字段； 数据库可能无法有效的对复杂的主键进行索引。 另一方面，对于一对多、多对多关联，特别是合成的标识符来说，集合也可以达到同样的高效性能。（ 附注：如果你希望 `SchemaExport` 为你的 `<set>` 创建主键， 你必须把所有的字段都声明为 `not-null="true"` 。）

`<idbag>` 映射定义了代理键，因此它总是可以很高效的被更新。事实上， `<idbag>` 拥有着最好的性能表现。

Bag是最差的。因为bag允许重复的元素值，也没有索引字段，因此不可能定义主键。 Hibernate无法判断出重复的行。当这种集合被更改时，Hibernate将会先完整地移除 （通过一个(in a single `DELETE`)）整个集合，然后再重新创建整个集合。因此Bag是非常低效的。

请注意：对于一对多关联来说，“主键”很可能并不是数据库表的物理主键。 但就算在此情况下，上面的分类仍然是有用的。（它仍然反映了Hibernate在集合的各数据行中是如何进行“定位”的。）

19.5.2. Lists, maps 和sets用于更新效率最高

根据我们上面的讨论，显然有序集合类型和大多数set都可以在增加、删除、修改元素中拥有最好的性能。

可论证的是对于多对多关联、值数据集合而言，有序集合类比集合(set)有一个好处。因为 `Set` 的内在结构， 如果“改变”了一个元素，Hibernate并不会 `更新（UPDATE）` 这一行。 对于 `Set` 来说，只有在 `插入（INSERT）` 和 `删除（DELETE）` 操作时“改变”才有效。再次强调：这段讨论对“一对多关联”并不适用。

注意到数组无法延迟载入，我们可以得出结论，list, map和idbags是最高效的（非反向）集合类型，set则紧随其后。 在Hibernate中，set应该时最通用的集合类型，这时因为“set”的语义在关系模型中是最自然的。

但是，在设计良好的Hibernate领域模型中，我们通常可以看到更多的集合事实上是带有 `inverse="true"` 的一对多的关联。对于这些关联，更新操作将会在多对一的这一端进行处理。因此对于此类情况，无需考虑其集合的更新性能。

19.5.3. Bag和list是反向集合类中效率最高的

在把bag扔进水沟之前，你必须了解，在一种情况下，bag的性能(包括list)要比set高得多： 对于指明了 `inverse="true"` 的集合类（比如说，标准的双向的一对多关联）， 我们可以在未初始化(fetch)包元素的情况下直接向bag或list添加新元素！ 这是因为 `Collection.add()`)或者 `Collection.addAll()` 方法 对bag或者List总是返回true（这点与与Set不同）。因此对于下面的相同代码来说，速度会快得多。

```
Parent p = (Parent) sess.load(Parent.class, id);
    Child c = new Child();
    c.setParent(p);
    p.getChildren().add(c);  //no need to fetch the collection!
    sess.flush();
```

19.5.4. 一次性删除（One shot delete）

偶尔的，逐个删除集合类中的元素是相当低效的。Hibernate并没那么笨， 如果你想要把整个集合都删除（比如说调用list.clear()），Hibernate只需要一个DELETE就搞定了。

假设我们在一个长度为20的集合类中新增加了一个元素，然后再删除两个。Hibernate会安排一条 `INSERT` 语句和两条 `DELETE` 语句（除非集合类是一个bag)。 这当然是显而易见的。

但是，假设我们删除了18个数据，只剩下2个，然后新增3个。则有两种处理方式：

- 逐一的删除这18个数据，再新增三个；
- 删除整个集合类（只用一句DELETE语句），然后增加5个数据。

Hibernate还没那么聪明，知道第二种选择可能会比较快。 （也许让Hibernate不这么聪明也是好事，否则可能会引发意外的“数据库触发器”之类的问题。）

幸运的是，你可以强制使用第二种策略。你需要取消原来的整个集合类（解除其引用）， 然后再返回一个新的实例化的集合类，只包含需要的元素。有些时候这是非常有用的。

显然，一次性删除并不适用于被映射为 `inverse="true"` 的集合。

19.6. 监测性能（Monitoring performance）

没有监测和性能参数而进行优化是毫无意义的。Hibernate为其内部操作提供了一系列的示意图，因此可以从 每个 `SessionFactory` 抓取其统计数据。

19.6.1. 监测SessionFactory

你可以有两种方式访问 `SessionFactory` 的数据记录，第一种就是自己直接调用 `sessionFactory.getStatistics()` 方法读取、显示 `统计` 数据。

此外，如果你打开 `StatisticsService` MBean选项，那么Hibernate则可以使用JMX技术 发布其数据记录。你可以让应用中所有的 `SessionFactory` 同时共享一个MBean，也可以每个 SessionFactory分配一个MBean。下面的代码即是其演示代码：

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object
 name

StatisticsService stats = new StatisticsService(); // MBean impl
ementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a
SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the se
rver
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object
 name

StatisticsService stats = new StatisticsService(); // MBean impl
ementation
server.registerMBean(stats, on); // Register the MBean on the se
rver
```

TODO：仍需要说明的是：在第一个例子中，我们直接得到和使用MBean；而在第二个例子中，在使用MBean之前 我们则需要给出SessionFactory的JNDI名，使用 `hibernateStatsBean.setSessionFactoryJNDIName("my/JNDI/Name")` 得到SessionFactory，然后将MBean保存于其中。

你可以通过以下方法打开或关闭 `SessionFactory` 的监测功能：

- 在配置期间，将 `hibernate.generate_statistics` 设置为 `true` 或 `false` ；

- 在运行期间，则可以可以通过 `sf.getStatistics().setStatisticsEnabled(true)` 或 `hibernateStatsBean.setStatisticsEnabled(true)`

你也可以在程序中调用 `clear()` 方法重置统计数据，调用 `logSummary()` 在日志中记录（info级别）其总结。

19.6.2. 数据记录（Metrics）

Hibernate提供了一系列数据记录，其记录的内容包括从最基本的信息到与具体场景的特殊信息。所有的测量值都可以由 `Statistics` 接口进行访问，主要分为三类：

- 使用 `Session` 的普通数据记录，例如打开的Session的个数、取得的JDBC的连接数等；
- 实体、集合、查询、缓存等内容的统一数据记录
- 和具体实体、集合、查询、缓存相关的详细数据记录

例如：你可以检查缓存的命中成功次数，缓存的命中失败次数，实体、集合和查询的使用概率，查询的平均时间等。请注意 Java中时间的近似精度是毫秒。Hibernate的数据精度和具体的JVM有关，在有些平台上其精度甚至只能精确到10秒。

你可以直接使用getter方法得到全局数据记录（例如，和具体的实体、集合、缓存区无关的数据），你也可以在具体查询中通过标记实体名、 或HQL、SQL语句得到某实体的数据记录。请参考 `Statistics` 、 `EntityStatistics` 、 `CollectionStatistics` 、 `SecondLevelCacheStatistics` 、和 `QueryStatistics` 的API文档以抓取更多信息。下面的代码则是个简单的例子：

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount
);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"
);
```

如果你想得到所有实体、集合、查询和缓存区的数据，你可以通过以下方法获得实体、集合、查询和缓存区列表： `getQueries()` 、 `getEntityNames()` 、 `getCollectionRoleNames()` 和 `getSecondLevelCacheRegionNames()` 。

第 20 章 工具箱指南

目录

可以通过一系列Eclipse插件、命令行工具和Ant任务来进行与Hibernate关联的转换。

除了Ant任务外，当前的*Hibernate Tools*也包含了Eclipse IDE的插件，用于与现存数据库的逆向工程。

- *Mapping Editor:* Hibernate XML映射文件的编辑器，支持自动完成和语法高亮。它也支持对类名和属性/字段名的语义自动完成，比通常的XML编辑器方便得多。

- *Console:* Console是Eclipse的一个新视图。除了对你的console配置的树状概览，你还可以获得对你持久化类及其关联的交互式视图。Console允许你对数据库执行HQL查询，并直接在Eclipse中浏览结果。

- *Development Wizards:* 在Hibernate Eclipse tools中还提供了几个向导；你可以用向导快速生成Hibernate 配置文件（cfg.xml），你甚至还可以同现存的数据库schema中反向工程出POJO源代码与Hibernate 映射文件。反向工程支持可定制的模版。

- *Ant Tasks:*

要得到更多信息，请查阅 *Hibernate Tools* 包及其文档。

同时，Hibernate主发行包还附带了一个集成的工具（它甚至可以在Hibernate“内部”快速运行）*SchemaExport* ，也就是 `hbm2ddl` 。

20.1. Schema自动生成（Automatic schema generation）

可以从你的映射文件使用一个Hibernate工具生成DDL。 生成的schema包含有对实体和集合类表的完整性引用约束（主键和外键）。涉及到的标示符生成器所需的表和sequence也会同时生成。

在使用这个工具的时候，你必须 通过`hibernate.dialet`属性指定一个SQL`方言(Dialet)`，因为DDL是与供应商高度相关的。

首先，要定制你的映射文件，来改善生成的schema。

20.1.1. 对schema定制化(Customizing the schema)

很多Hibernate映射元素定义了可选的 `length` 、 `precision` 或者 `scale` 属性。你可以通过这个属性设置字段的长度、精度、小数点位数。

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

有些tag还接受 `not-null` 属性（用来在表字段上生成 `NOT NULL` 约束）和 `unique` 属性（用来在表字段上生成 `UNIQUE` 约束）。

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" uniqu
e="true"/>
```

`unique-key` 属性可以对成组的字段指定一个唯一键约束(unique key constraint)。目前， `unique-key` 属性指定的值在生成DDL时并不会被当作这个约束的名字，它们只是在用来在映射文件内部用作区分的。

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId
"/>
<property name="employeeId" unique-key="OrgEmployee"/>
```

`index` 属性会用对应的字段（一个或多个）生成一个index,它指出了这个index的名字。如果多个字段对应的index名字相同，就会生成包含这些字段的index。

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

`foreign-key` 属性可以用来覆盖任何生成的外键约束的名字。

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

很多映射元素还接受 `<column>` 子元素。这在定义跨越多字段的类型时特别有用。

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="
30"/>
    <column name="first" not-null="true" index="bar_idx" length=
"20"/>
    <column name="initial"/>
</property>
```

`default` 属性为字段指定一个默认值 (在保存被映射的类的新实例之前，你应该将同样的值赋于对应的属性)。

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</property>
```

`sql-type` 属性允许用户覆盖默认的Hibernate类型到SQL数据类型的映射。

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property>
```

`check` 属性允许用户指定一个约束检查。

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

表 **20.1. Summary**

属性(Attribute)	值（Values）	解释（Interpretatior
`length`	数字	字段长度
`precision`	数字	精度(decimal precision)
`scale`	数字	小数点位数(decimal scale)
`not-null`	`true|false`	指明字段是否应该是非空的
`unique`	`true|false`	指明是否该字段具有惟一约束
`index`	`index_name`	指明一个（多字段）的索引(index)的
`unique-key`	`unique_key_name`	指明多字段惟一约束的名字（参见上
`foreign-key`	`foreign_key_name`	specifies the name of the foreign key generated for an association, for a `<one-to-one>` , `<many` `<key>` , or `<many-to-m` mapping element. Note that `invers` will not be considered by `SchemaEx` 外键的名字，它是为关联生成的，或者 `<one-to-one>` ， `<ma` `<key>` ，或者 `<many-to-` 元素。注意 `inverse="true"` 在 `Sc` 被忽略。
`sql-type`	`SQL 字段类型`	覆盖默认的字段类型(只能用于 `<` 性）
`default`	SQL表达式	为字段指定默认值
`check`	SQL 表达式	对字段或表加入SQL约束检查

`<comment>` 元素可以让你在生成的schema中加入注释。

```
<class name="Customer" table="CurCust">
    <comment>Current customers only</comment>
    ...
</class>
```

```
<property name="balance">
    <column name="bal">
        <comment>Balance in USD</comment>
    </column>
</property>
```

结果是在生成的DDL中包含`comment on table` 或者 `comment on column`语句(假若支持的话)。

20.1.2. 运行该工具

`SchemaExport` 工具把DDL脚本写到标准输出，同时/或者执行DDL语句。

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaExport` *options mapping_files*

表 **20.2.** **`SchemaExport`** 命令行选项

选项	说明
`--quiet`	不要把脚本输出到stdout
`--drop`	只进行drop tables的步骤
`--create`	只创建表
`--text`	不执行在数据库中运行的步骤
`--output=my_schema.ddl`	把输出的ddl脚本输出到一个文件
`--naming=eg.MyNamingStrategy`	选择一个命名策略（ `NamingStrategy` ）
`--config=hibernate.cfg.xml`	从XML文件读入Hibernate配置
`--properties=hibernate.properties`	从文件读入数据库属性
`--format`	把脚本中的SQL语句对齐和美化
`--delimiter=;`	为脚本设置行结束符

你甚至可以在你的应用程序中嵌入 `SchemaExport` 工具:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

20.1.3. 属性(Properties)

可以通过如下方式指定数据库属性:

- 通过 `-D` *<property>*系统参数
- 在 `hibernate.properties` 文件中
- 位于一个其它名字的properties文件中,然后用 `--properties` 参数指定

所需的参数包括:

表 **20.3. SchemaExport** 连接属性

属性名	说明
`hibernate.connection.driver_class`	jdbc driver class
`hibernate.connection.url`	jdbc url
`hibernate.connection.username`	database user
`hibernate.connection.password`	user password
`hibernate.dialect`	方言(dialect)

20.1.4. 使用Ant(Using Ant)

你可以在你的Ant build脚本中调用 `SchemaExport` ：

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

20.1.5. 对schema的增量更新(Incremental schema updates)

`SchemaUpdate` 工具对已存在的schema采用"增量"方式进行更新。注意 `SchemaUpdate` 严重依赖于JDBC metadata API,所以它并非对所有JDBC驱动都有效。

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaUpdate` *options mapping_files*

表 **20.4.** **`SchemaUpdate`** 命令行选项

选项	说明
`--quiet`	不要把脚本输出到stdout
`--text`	不把脚本输出到数据库
`--naming=eg.MyNamingStrategy`	选择一个命名策略 (`NamingStrategy`)
`--properties=hibernate.properties`	从指定文件读入数据库属性
`--config=hibernate.cfg.xml`	指定一个 `.cfg.xml` 文件

你可以在你的应用程序中嵌入 `SchemaUpdate` 工具:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

20.1.6. 用Ant来增量更新schema(Using Ant for incremental schema updates)

你可以在Ant脚本中调用 `SchemaUpdate` ：

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

20.1.7. Schema 校验

`SchemaValidator` 工具会比较数据库现状是否与映射文档“匹配”。注意，`SchemaValidator` 严重依赖于JDBC的metadata API，因此不是对所有的JDBC驱动都适用。这一工具在测试的时候特别有用。

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaValidator` *options mapping_files*

表 **20.5.** **`SchemaValidator`** 命令行参数

选项	描述
`--naming=eg.MyNamingStrategy`	选择一个命名策略 (`NamingStrategy`)
`--properties=hibernate.properties`	从文件中读取数据库属性
`--config=hibernate.cfg.xml`	指定一个 `.cfg.xml` 文件

你可以在你的应用程序中嵌入 `SchemaValidator` ：

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

20.1.8. 使用Ant进行schema校验

你可以在Ant脚本中调用 `SchemaValidator`：

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTas
k"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

第 21 章 示例：父子关系(Parent Child Relationships)

目录

刚刚接触Hibernate的人大多是从父子关系（parent / child type relationship）的建模入手的。父子关系的建模有两种方法。由于种种原因，最方便的方法是把 `Parent` 和 `Child` 都建模成实体类，并创建一个从 `Parent` 指向 `Child` 的<one-to-many>关联，对新手来说尤其如此。还有一种方法，就是将 `Child` 声明为一个 `<composite-element>` （组合元素）。 事实上在Hibernate中one to many关联的默认语义远没有composite element贴近parent / child关系的通常语义。下面我们会阐述如何使用带有级联的双向一对多关联*(bidirectional one to many association with cascades)*去建立有效、优美的parent / child关系。这一点也不难！

21.1. 关于collections需要注意的一点

Hibernate collections被当作其所属实体而不是其包含实体的一个逻辑部分。这非常重要！它主要体现为以下几点：

- 当删除或增加collection中对象的时候，collection所属者的版本值会递增。
- 如果一个从collection中移除的对象是一个值类型(value type)的实例，比如composite element，那么这个对象的持久化状态将会终止，其在数据库中对应的记录会被删除。同样的，向collection增加一个value type的实例将会使之立即被持久化。
- 另一方面，如果从一对多或多对多关联的collection中移除一个实体，在缺省情况下这个对象并不会被删除。这个行为是完全合乎逻辑的－－改变一个实体的内部状态不应该使与它关联的实体消失掉！同样的，向collection增加一个实体不会使之被持久化。

实际上，向Collection增加一个实体的缺省动作只是在两个实体之间创建一个连接而已，同样移除的时候也只是删除连接。这种处理对于所有的情况都是合适的。对于父子关系则是完全不适合的，在这种关系下，子对象的生存绑定于父对象的生存周期。

21.2. 双向的一对多关系(Bidirectional one-to-many)

假设我们要实现一个简单的从Parent到Child的<one-to-many>关联。

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

如果我们运行下面的代码

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate会产生两条SQL语句：

- 一条 `INSERT` 语句，为 `c` 创建一条记录
- 一条 `UPDATE` 语句，创建从 `p` 到 `c` 的连接

这样做不仅效率低，而且违反了列 `parent_id` 非空的限制。我们可以通过在集合类映射上指定 `not-null="true"` 来解决违反非空约束的问题：

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set>
```

然而，这并非是推荐的解决方法。

这种现象的根本原因是从 `p` 到 `c` 的连接（外键parent_id）没有被当作 `Child` 对象状态的一部分，因而没有在INSERT语句中被创建。因此解决的办法就是把这个连接添加到Child的映射中。

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

（我们还需要为类 `Child` 添加 `parent` 属性）

现在实体 `Child` 在管理连接的状态，为了使collection不更新连接，我们使用 `inverse` 属性。

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

下面的代码是用来添加一个新的 `Child`

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

现在，只会有一条 `INSERT` 语句被执行！

为了让事情变得井井有条，可以为 `Parent` 加一个 `addChild()` 方法。

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

现在，添加 `Child` 的代码就是这样

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

21.3. 级联生命周期（Cascading lifecycle）

需要显式调用 `save()` 仍然很麻烦，我们可以用级联来解决这个问题。

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

这样上面的代码可以简化为：

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

同样的，保存或删除 `Parent` 对象的时候并不需要遍历其子对象。 下面的代码会删除对象 `p` 及其所有子对象对应的数据库记录。

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

然而，这段代码

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

不会从数据库删除 `c` ；它只会删除与 `p` 之间的连接（并且会导致违反 `NOT NULL` 约束，在这个例子中）。你需要显式调用 `delete()` 来删除 `Child` 。

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

在我们的例子中，如果没有父对象，子对象就不应该存在，如果将子对象从collection中移除，实际上我们是想删除它。要实现这种要求，就必须使用 `cascade="all-delete-orphan"` 。

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

注意：即使在collection一方的映射中指定 `inverse="true"` ，级联仍然是通过遍历collection中的元素来处理的。如果你想要通过级联进行子对象的插入、删除、更新操作，就必须把它加到collection中，只调用 `setParent()` 是不够的。

21.4. 级联与 未保存值 （Cascades and unsaved-value ）

假设我们从 `Session` 中装入了一个 `Parent` 对象，用户界面对其进行了修改，然后希望在一个新的Session里面调用 `update()` 来保存这些修改。对象 `Parent` 包含了子对象的集合，由于打开了级联更新，Hibernate需要知道哪些Child对象是新实例化的，哪些代表数据库中已经存在的记录。我们假设 `Parent` 和 `Child` 对象的标识属性都是自动生成的，类型为 `java.lang.Long` 。Hibernate会使用标识属性的值，和version 或 timestamp 属性，来判断哪些子对象是新的。(参见第 10.7 节 "自动状态检测".) 在 *Hibernate3* 中,显式指定 *`unsaved-value`* 不再是必须的了。

下面的代码会更新 `parent` 和 `child` 对象，并且插入 `newChild` 对象。

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Well, that's all very well for the case of a generated identifier, but what about assigned identifiers and composite identifiers? This is more difficult, since Hibernate can't use the identifier property to distinguish between a newly instantiated object (with an identifier assigned by the user) and an object loaded in a previous session. In this case, Hibernate will either use the timestamp or version property, or will actually query the second-level cache or, worst case, the database, to see if the row exists.

这对于自动生成标识的情况是非常好的，但是自分配的标识和复合标识怎么办呢？这是有点麻烦，因为Hibernate没有办法区分新实例化的对象（标识被用户指定了）和前一个Session装入的对象。在这种情况下，Hibernate会使用timestamp或version属性，或者查询第二级缓存，或者最坏的情况，查询数据库，来确认是否此行存在。

21.5. 结论

这里有不少东西需要融会贯通，可能会让新手感到迷惑。但是在实践中它们都工作地非常好。大部分Hibernate应用程序都会经常用到父子对象模式。

在第一段中我们曾经提到另一个方案。上面的这些问题都不会出现在 <composite-element> 映射中，它准确地表达了父子关系的语义。很不幸复合元素还有两个重大限制:复合元素不能拥有collections，并且，除了用于惟一的父对象外，它们不能再作为其它任何实体的子对象。

第 22 章 示例：Weblog 应用程序

目录

22.1. 持久化类

下面的持久化类表示一个weblog和在其中张贴的一个贴子。他们是标准的父/子关系模型，但是我们会用一个有序包（ordered bag)而非集合(set)。

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

22.2. Hibernate 映射

下列的XML映射应该是很直白的。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.
dtd">

<hibernate-mapping package="eg">
    <class
        name="Blog"
        table="BLOGS" >

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.
dtd">

<hibernate-mapping package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

22.3. Hibernate 代码

下面的类演示了我们可以使用Hibernate对这些类进行的一些操作。

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateExceptio
n {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
```

```
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, Stri
ng text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, St
ring text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
```

```
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.cla
ss, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
```

```
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.uniqueResult();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
```

```
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateExcept
ion {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

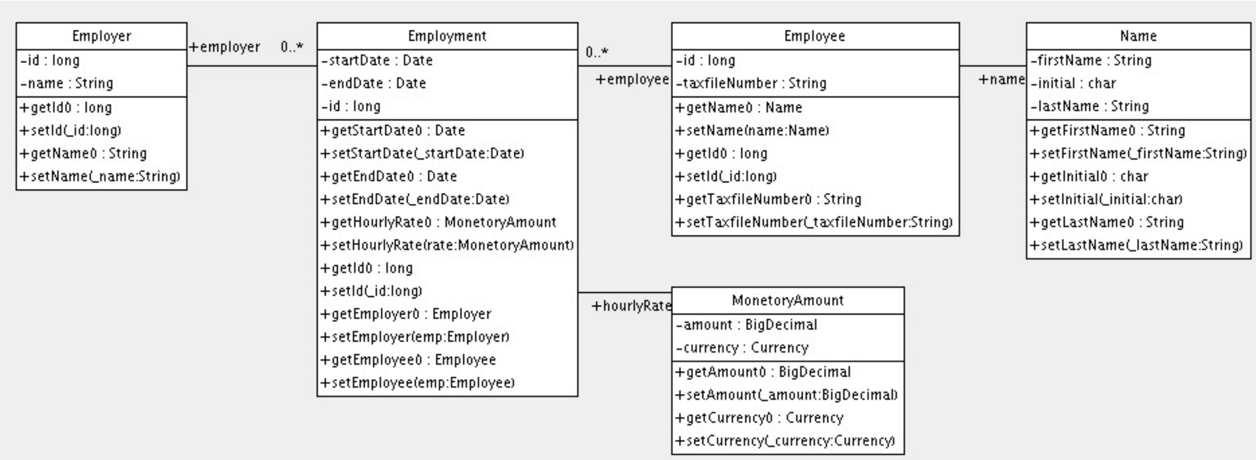
第 23 章 示例：复杂映射实例

目录

本章展示了一些较为复杂的关系映射。

23.1. Employer（雇主)/Employee(雇员)

下面关于 `Employer` 和 `Employee` 的关系模型使用了一个真实的实体类(`Employment`)来表述，这是因为对于相同的雇员和雇主可能会有多个雇佣时间段。 对于金额和雇员姓名，用Components建模。



映射文件可能是这样:

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetaryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12,
 2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-nu
ll="true"/>
        <many-to-one name="employee" column="employee_id" not-nu
ll="true"/>

    </class>

    <class name="Employee" table="employees">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

用 `SchemaExport` 生成表结构。

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

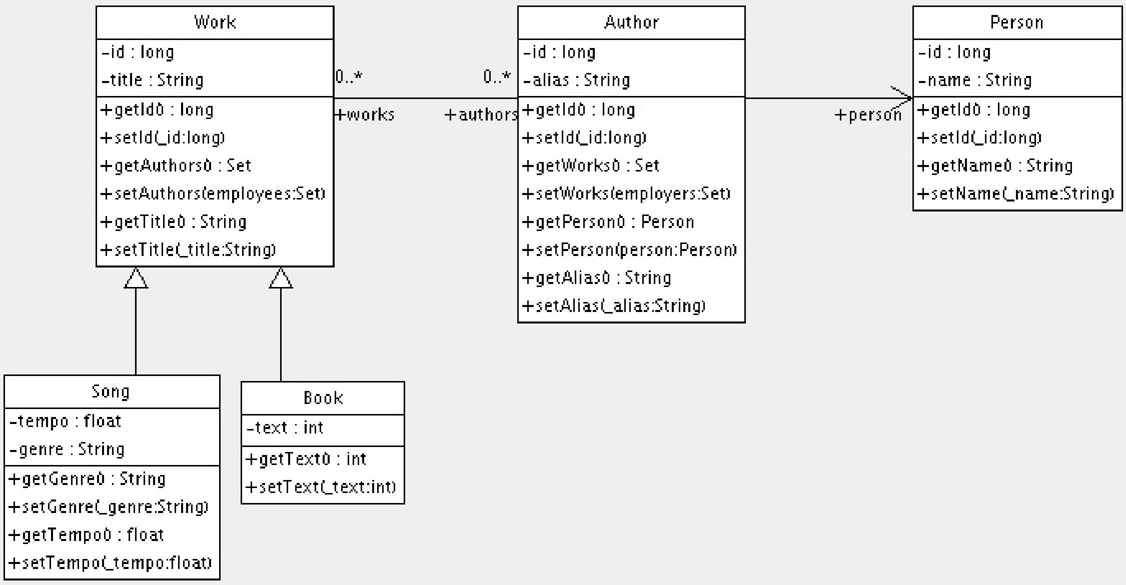
create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_i
d) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_i
d) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

23.2. Author(作家)/Work(作品)

考虑下面的 `Work`, `Author` 和 `Person` 模型的关系。我们用多对多关系来描述 `Work` 和 `Author`，用一对一关系来描述 `Author` 和 `Person`，另一种可能性是 `Author` 继承 `Person`。



下面的映射文件正确的描述了这些关系：

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"
/>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
```

```
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the
 Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

映射中有4个表。 `works` , `authors` 和 `persons` 分别保存着work，author和person的数据。 `author_work` 是authors和works的关联表。 表结构是由 `SchemaExport` 生成的。

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

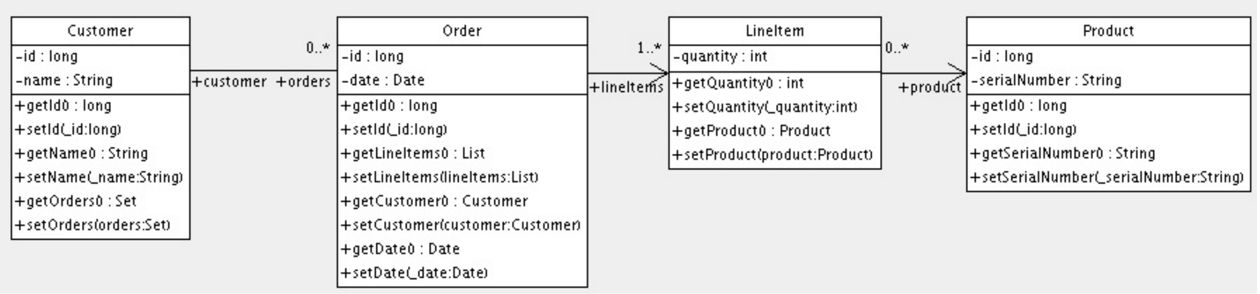
create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references person
s
alter table author_work
    add constraint author_workFK0 foreign key (author_id) refere
nces authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) referenc
es works
```

23.3. Customer(客户)/Order(订单)/Product(产品)

现在来考虑 `Customer`，`Order` ， `LineItem` 和 `Product` 关系的模型。 `Customer` 和 `Order` 之间 是一对多的关系，但是我们怎么来描述 `Order` / `LineItem` / `Product` 呢？ 我可以把 `LineItem` 作为描述 `Order` 和 `Product` 多对多关系的关联类，在Hibernate，这叫做组合元素。



映射文件如下：

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items">
            <key column="order_id"/>
            <list-index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/
>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

`customers` , `orders` , `line_items` 和 `products` 分别保存着customer, order, order line item 和 product的数据。 `line_items` 也作为连接orders 和 products的关联表。

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) reference
s customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) refere
nces products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) referenc
es orders
```

23.4. 杂例

这些例子全部来自于Hibernate的test suite，同时你也可以找到其他有用的例子。可以参考Hibernate的 `test` 目录。

TODO: put words around this stuff

23.4.1. "Typed" one-to-one association

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula>name</formula>
        <formula>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula>name</formula>
        <formula>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class>
```

23.4.2. Composite key example

```
<class name="Customer">

    <id name="customerId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="name" not-null="true" length="100"/>
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
```

```
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>

</class>

<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>

    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
```

```
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class>
```

23.4.3. 共有组合键属性的多对多(Many-to-many with shared composite key attribute)

```
<class name="User" table="`User`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>

<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
        <many-to-many class="User">
            <column name="userName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>
```

23.4.4. Content based discrimination

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>

    <discriminator
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>

    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class>
```

23.4.5. Associations on alternate keys

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>

    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>

    <property name="userId" length="8"/>

</class>

<class name="Address">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class>
```

第 24 章 最佳实践(Best Practices)

设计细颗粒度的持久类并且使用 `<component>` 来实现映射。

使用一个 `Address` 持久类来封装 `street` , `suburb` , `state` , `postcode` . 这将有利于代码重用和简化代码重构(refactoring)的工作。

对持久类声明标识符属性(identifier properties)。

Hibernate中标识符属性是可选的，不过有很多原因来说明你应该使用标识符属性。我们建议标识符应该是“人造”的(自动生成，不涉及业务含义)。

使用自然键(natural keys)标识

对所有的实体都标识出自然键，用 `<natural-id>` 进行映射。实现 `equals()` 和 `hashCode()` ，在其中用组成自然键的属性进行比较。

为每个持久类写一个映射文件

不要把所有的持久类映射都写到一个大文件中。把 `com.eg.Foo` 映射到 `com/eg/Foo.hbm.xml` 中， 在团队开发环境中，这一点显得特别有意义。

把映射文件作为资源加载

把映射文件和他们的映射类放在一起进行部署。

考虑把查询字符串放在程序外面

如果你的查询中调用了非ANSI标准的SQL函数，那么这条实践经验对你适用。把查询字符串放在映射文件中可以让程序具有更好的可移植性。

使用绑定变量

就像在JDBC编程中一样，应该总是用占位符"?"来替换非常量值，不要在查询中用字符串值来构造非常量值！更好的办法是在查询中使用命名参数。

不要自己来管理JDBC connections

Hibernate允许应用程序自己来管理JDBC connections，但是应该作为最后没有办法的办法。如果你不能使用Hibernate内建的connections providers，那么考虑实现自己来实现 `org.hibernate.connection.ConnectionProvider`

考虑使用用户自定义类型(custom type)

假设你有一个Java类型，来自某些类库，需要被持久化，但是该类没有提供映射操作需要的存取方法。那么你应该考虑实现 `org.hibernate.UserType` 接口。这种办法使程序代码写起来更加自如，不再需要考虑类与Hibernate type之间的相互转换。

在性能瓶颈的地方使用硬编码的JDBC

In performance-critical areas of the system, some kinds of operations might benefit from direct JDBC. But please, wait until you *know* something is a bottleneck. And don't assume that direct JDBC is necessarily faster. If you need to use direct JDBC, it might be worth opening a Hibernate `Session` and using that JDBC connection. That way you can still use the same transaction strategy and underlying connection provider. 在系统中对性能要求很严格的一些部分，某些操作也许直接使用JDBC会更好。但是请先确认这的确是一个瓶颈，并且不要想当然认为JDBC一定会更快。如果确实需要直接使用JDBC，那么最好打开一个 Hibernate `Session` 然后从 `Session` 获得connection，按照这种办法你仍然可以使用同样的transaction策略和底层的connection provider。

理解 `Session` 清洗（ flushing）

Session会不时的向数据库同步持久化状态，如果这种操作进行的过于频繁，性能会受到一定的影响。有时候你可以通过禁止自动flushing，尽量最小化非必要的flushing操作，或者更进一步，在一个特定的transaction中改变查询和其它操作的顺序。

在三层结构中，考虑使用托管对象（detached object）

当使用一个servlet / session bean 类型的架构的时候, 你可以把已加载的持久对象在session bean层和servlet / JSP 层之间来回传递。使用新的session来为每个请求服务，使用 `Session.merge()` 或者 `Session.saveOrUpdate()` 来与数据库同步。

在两层结构中，考虑使用长持久上下文(long persistence contexts).

为了得到最佳的可伸缩性，数据库事务(Database Transaction)应该尽可能的短。但是，程序常常需要实现长时间运行的“应用程序事务*(Application Transaction)”*，包含一个从用户的观点来看的原子操作。这个应用程序事务可能跨越多次从用户请求到得到反馈的循环。用脱管对象(与session脱离的对象)来实现应用程序事务是常见的。或者，尤其在两层结构中，把Hibernate Session从JDBC连接中脱离开，下次需要用的时候再连接上。绝不要把一个Session用在多个应用程序事务(Application Transaction)中，否则你的数据可能会过期失效。

不要把异常看成可恢复的

这一点甚至比“最佳实践”还要重要，这是“必备常识”。当异常发生的时候，必须要回滚 `Transaction` ，关闭 `Session` 。如果你不这样做的话，Hibernate无法保证内存状态精确的反应持久状态。尤其不要使用 `Session.load()` 来判断一个给定标识符的对象实例在数据库中是否存在，应该使用 `Session.get()` 或者进行一次查询.

对于关联优先考虑lazy fetching

谨慎的使用主动抓取(eager fetching)。对于关联来说，若其目标是无法在第二级缓存中完全缓存所有实例的类，应该使用代理(proxies)与/或具有延迟加载属性的集合(lazy collections)。若目标是可以被缓存的，尤其是缓存的命中率非常高的情况下，应该使用 `lazy="false"` ，明确的禁止掉eager fetching。如果那些特殊的确实适合使用join fetch 的场合，请在查询中使用 `left join fetch` 。

使用*open session in view*模式，或者执行严格的装配期*(assembly phase)*策略来避免再次抓取数据带来的问题

Hibernate让开发者们摆脱了繁琐的*Data Transfer Objects* (DTO)。在传统的EJB结构中，DTO有双重作用：首先，他们解决了entity bean无法序列化的问题；其次，他们隐含地定义了一个装配期，在此期间，所有在view层需要用到的数据，都被抓取、集中到了DTO中，然后控制才被装到表示层。Hibernate终结了第一个作用。然而，除非你做好了在整个渲染过程中都维护一个打开的持久化上下文(session)的准备，你仍然需要一个装配期（想象一下，你的业务方法与你的表示层有严格的契约，数据总是被放置到托管对象中）。这并非是Hibernate的限制！这是实现安全的事务化数据访问的基本需求。

考虑把Hibernate代码从业务逻辑代码中抽象出来

把Hibernate的数据存取代码隐藏到接口(interface)的后面，组合使用*DAO*和*Thread Local Session*模式。通过Hibernate的 `UserType` ，你甚至可以用硬编码的JDBC来持久化那些本该被Hibernate持久化的类。 (该建议更适用于规模足够大应用软件中，对于那些只有5张表的应用程序并不适合。)

不要用怪异的连接映射

多对多连接用得好的例子实际上相当少见。大多数时候你在“连接表”中需要保存额外的信息。这种情况下，用两个指向中介类的一对多的连接比较好。实际上，我们认为绝大多数的连接是一对多和多对一的，你应该谨慎使用其它连接风格，用之前问自己一句，是否真的必须这么做。

偏爱双向关联

单向关联更加难于查询。在大型应用中，几乎所有的关联必须在查询中可以双向导航。

HttpClient 教程

南磊 译

前言

超文本传输协议（HTTP）也许是当今互联网上使用的最重要的协议了。Web服务，有网络功能的设备和网络计算的发展，都持续扩展了HTTP协议的角色，超越了用户使用的Web浏览器范畴，同时，也增加了需要HTTP协议支持的应用程序的数量。

尽管java.net包提供了基本通过HTTP访问资源的功能，但它没有提供全面的灵活性和其它很多应用程序需要的功能。HttpClient就是寻求弥补这项空白的组件，通过提供一个有效的，保持更新的，功能丰富的软件包来实现客户端最新的HTTP标准和建议。

为扩展而设计，同时为基本的HTTP协议提供强大的支持，HttpClient组件也许就是构建HTTP客户端应用程序，比如web浏览器，web服务端，利用或扩展HTTP协议进行分布式通信的系统的开发人员的关注点。

1. HttpClient的范围

- 基于HttpCore的客户端HTTP运输实现库
- 基于经典（阻塞）I/O
- 内容无关

2. 什么是HttpClient不能做的

- HttpClient不是一个浏览器。它是一个客户端的HTTP通信实现库。HttpClient的目标是发送和接收HTTP报文。HttpClient不会去缓存内容，执行嵌入在HTML页面中的javascript代码，猜测内容类型，重新格式化请求/重定向URI，或者其它和HTTP运输无关的功能。

关于翻译

本文档翻译工作由南磊完成，版权归译者所有。免费发布，但是不可擅自用于任何与商 业有关的用途。若对翻译质量有任何意见或建议，可以联系译者 nanlei1987@gmail.com 。 英文原版由 Oleg Kalnichevski 所著，若对 HttpClient 本身有问题的可以直接反馈到 Apache 官 网 HttpClient 项目组

第一章 基础

1.1 执行请求

HttpClient最重要的功能是执行HTTP方法。一个HTTP方法的执行包含一个或多个HTTP请求/HTTP响应交换，通常由HttpClient的内部来处理。而期望用户提供一个要执行的请求对象，而HttpClient期望传输请求到目标服务器并返回对应的响应对象，或者当执行不成功时抛出异常。

很自然地，HttpClient API的主要切入点就是定义描述上述规约的HttpClient接口。

这里有一个很简单的请求执行过程的示例：

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    InputStream instream = entity.getContent();
    int l;
    byte[] tmp = new byte[2048];
    while ((l = instream.read(tmp)) != -1) {
    }
}
```

1.1.1 HTTP请求

所有HTTP请求有一个组合了方法名，请求URI和HTTP协议版本的请求行。

HttpClient支持所有定义在HTTP/1.1版本中的HTTP方法：GET，HEAD，POST，PUT，DELETE，TRACE和OPTIONS。对于每个方法类型都有一个特殊的类：HttpGet，HttpHead，HttpPost，HttpPut，HttpDelete，HttpTrace和HttpOptions。

请求的URI是统一资源定位符，它标识了应用于哪个请求之上的资源。HTTP请求URI包含一个协议模式，主机名称，可选的端口，资源路径，可选的查询和可选的片段。

```
HttpGet httpget = new HttpGet(
    "http://www.google.com/search?hl=en&q=httpclient&btnG=Google
+Search&aq=f&oq=");
```

HttpClient提供很多工具方法来简化创建和修改执行URI。 URI也可以编程来拼装：

```
URI uri = URIUtils.createURI("http", "www.google.com", -1, "/sea
rch",
    "q=httpclient&btnG=Google+Search&aq=f&oq=", null);
HttpGet httpget = new HttpGet(uri);
System.out.println(httpget.getURI());
```

输出内容为：

```
http://www.google.com/search?q=httpclient&btnG=Google+Search&aq=
f&oq=
```

查询字符串也可以从独立的参数中来生成：

```
List<NameValuePair> qparams = new ArrayList<NameValuePair>();
qparams.add(new BasicNameValuePair("q", "httpclient"));
qparams.add(new BasicNameValuePair("btnG", "Google Search"));
qparams.add(new BasicNameValuePair("aq", "f"));
qparams.add(new BasicNameValuePair("oq", null));
URI uri = URIUtils.createURI("http", "www.google.com", -1, "/sea
rch",
URLEncodedUtils.format(qparams, "UTF-8"), null);
HttpGet httpget = new HttpGet(uri);
System.out.println(httpget.getURI());
```

输出内容为：

```
http://www.google.com/search?q=httpclient&btnG=Google+Search&aq=
f&oq=
```

1.1.2 HTTP响应

HTTP响应是由服务器在接收和解释请求报文之后返回发送给客户端的报文。响应报文的第一行包含了协议版本，之后是数字状态码和相关联的文本段。

```
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1
_1,
    HttpStatus.SC_OK, "OK");
System.out.println(response.getProtocolVersion());
System.out.println(response.getStatusLine().getStatusCode());
System.out.println(response.getStatusLine().getReasonPhrase());
System.out.println(response.getStatusLine().toString());
```

输出内容为：

```
HTTP/1.1
200
OK
HTTP/1.1 200 OK
```

1.1.3 处理报文头部

一个HTTP报文可以包含很多描述如内容长度，内容类型等信息属性的头部信息。

HttpClient提供获取，添加，移除和枚举头部信息的方法。

```
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
    HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie",
    "c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie",
    "c2=b; path=\"/\", c3=c; domain=\"localhost\"");
Header h1 = response.getFirstHeader("Set-Cookie");
System.out.println(h1);
Header h2 = response.getLastHeader("Set-Cookie");
System.out.println(h2);
Header[] hs = response.getHeaders("Set-Cookie");
System.out.println(hs.length);
```

输出内容为：

```
Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
```

获得给定类型的所有头部信息最有效的方式是使用HeaderIterator接口。

```
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1_1,
    HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie",
    "c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie",
    "c2=b; path=\"/\", c3=c; domain=\"localhost\"");
HeaderIterator it = response.headerIterator("Set-Cookie");
while (it.hasNext()) {
    System.out.println(it.next());
}
```

输出内容为：

```
Set-Cookie: c1=a; path=/; domain=localhost
Set-Cookie: c2=b; path="/", c3=c; domain="localhost"
```

它也提供解析HTTP报文到独立头部信息元素的方法方法。

```
HttpResponse response = new BasicHttpResponse(HttpVersion.HTTP_1
_1,
    HttpStatus.SC_OK, "OK");
response.addHeader("Set-Cookie",
    "c1=a; path=/; domain=localhost");
response.addHeader("Set-Cookie",
    "c2=b; path=\"/\", c3=c; domain=\"localhost\"");
HeaderElementIterator it = new BasicHeaderElementIterator(
    response.headerIterator("Set-Cookie"));
while (it.hasNext()) {
    HeaderElement elem = it.nextElement();
    System.out.println(elem.getName() + " = " + elem.getValue())
;
    NameValuePair[] params = elem.getParameters();
    for (int i = 0; i < params.length; i++) {
        System.out.println(" " + params[i]);
    }
}
```

输出内容为：

```
c1 = a
path=/
domain=localhost
c2 = b
path=/
c3 = c
domain=localhost
```

1.1.4 HTTP实体

HTTP报文可以携带和请求或响应相关的内容实体。实体可以在一些请求和响应中找到，因为它们也是可选的。使用了实体的请求被称为封闭实体请求。HTTP规范定义了两种封闭实体的方法：POST和PUT。响应通常期望包含一个内容实体。这个规则也有特例，比如HEAD方法的响应和204 No Content，304 Not Modified和205 Reset Content响应。

HttpClient根据其内容出自何处区分三种类型的实体：

- streamed流式：内容从流中获得，或者在运行中产生。特别是这种分类包含从

HTTP响应中获取的实体。流式实体是不可重复生成的。
- self-contained自我包含式：内容在内存中或通过独立的连接或其它实体中获得。自我包含式的实体是可以重复生成的。这种类型的实体会经常用于封闭HTTP请求的实体。
- wrapping包装式：内容从另外一个实体中获得。

当从一个HTTP响应中获取流式内容时，这个区别对于连接管理很重要。对于由应用程序创建而且只使用HttpClient发送的请求实体，流式和自我包含式的不同就不那么重要了。这种情况下，建议考虑如流式这种不能重复的实体，和可以重复的自我包含式实体。

1.1.4.1 重复实体

实体可以重复，意味着它的内容可以被多次读取。这就仅仅是自我包含式的实体了（像ByteArrayEntity或StringEntity）。

1.1.4.2 使用HTTP实体

因为一个实体既可以代表二进制内容又可以代表字符内容，它也支持字符编码（支持后者也就是字符内容）。

实体是当使用封闭内容执行请求，或当请求已经成功执行，或当响应体结果发功到客户端时创建的。

要从实体中读取内容，可以通过HttpEntity#getContent()方法从输入流中获取，这会返回一个java.io.InputStream对象，或者提供一个输出流到HttpEntity#writeTo(OutputStream)方法中，这会一次返回所有写入到给定流中的内容。

当实体通过一个收到的报文获取时，HttpEntity#getContentType()方法和HttpEntity#getContentLength()方法可以用来读取通用的元数据，如Content-Type和Content-Length头部信息（如果它们是可用的）。因为头部信息Content-Type可以包含对文本MIME类型的字符编码，比如text/plain或text/html，HttpEntity#getContentEncoding()方法用来读取这个信息。如果头部信息不可用，那么就返回长度-1，而对于内容类型返回NULL。如果头部信息Content-Type是可用的，那么就会返回一个Header对象。

当为一个传出报文创建实体时，这个元数据不得不通过实体创建器来提供。

```
StringEntity myEntity = new StringEntity("important message",
    "UTF-8");
System.out.println(myEntity.getContentType());
System.out.println(myEntity.getContentLength());
System.out.println(EntityUtils.getContentCharSet(myEntity));
System.out.println(EntityUtils.toString(myEntity));
System.out.println(EntityUtils.toByteArray(myEntity).length);
```

输出内容为

Content-Type: text/plain; charset=UTF-8 17 UTF-8 important message 17

1.1.5 确保低级别资源释放

当完成一个响应实体，那么保证所有实体内容已经被完全消耗是很重要的，所以连接可以安全的放回到连接池中，而且可以通过连接管理器对后续的请求重用连接。处理这个操作的最方便的方法是调用HttpEntity#consumeContent()方法来消耗流中的任意可用内容。HttpClient探测到内容流尾部已经到达后，会立即会自动释放低层连接，并放回到连接管理器。HttpEntity#consumeContent()方法调用多次也是安全的。

也可能会有特殊情况，当整个响应内容的一小部分需要获取，消耗剩余内容而损失性能，还有重用连接的代价太高，则可以仅仅通过调用HttpUriRequest#abort()方法来中止请求。

```
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    InputStream instream = entity.getContent();
    int byteOne = instream.read();
    int byteTwo = instream.read();
    // Do not need the rest
    httpget.abort();
}
```

连接不会被重用，但是由它持有的所有级别的资源将会被正确释放。

1.1.6 消耗实体内容

推荐消耗实体内容的方式是使用它的HttpEntity#getContent()或HttpEntity#writeTo(OutputStream)方法。HttpClient也自带EntityUtils类，这会暴露出一些静态方法，这些方法可以更加容易地从实体中读取内容或信息。代替直接读取java.io.InputStream，也可以使用这个类中的方法以字符串/字节数组的形式获取整个内容体。然而，EntityUtils的使用是强烈不鼓励的，除非响应实体源自可靠的HTTP服务器和已知的长度限制。

```
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    long len = entity.getContentLength();
    if (len != -1 && len < 2048) {
        System.out.println(EntityUtils.toString(entity));
    } else {
        // Stream content out
    }
}
```

在一些情况下可能会不止一次的读取实体。此时实体内容必须以某种方式在内存或磁盘上被缓冲起来。最简单的方法是通过使用BufferedHttpEntity类来包装源实体完成。这会引起源实体内容被读取到内存的缓冲区中。在其它所有方式中，实体包装器将会得到源实体。

```
HttpGet httpget = new HttpGet("http://localhost/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
if (entity != null) {
    entity = new BufferedHttpEntity(entity);
}
```

1.1.7 生成实体内容

HttpClient提供一些类，它们可以用于生成通过HTTP连接获得内容的有效输出流。为了封闭实体从HTTP请求中获得的输出内容，那些类的实例可以和封闭如POST和PUT请求的实体相关联。HttpClient为很多公用的数据容器，比如字符串，字节数组，输入流和文件提供了一些类：StringEntity，ByteArrayEntity，InputStreamEntity和FileEntity。

```
File file = new File("somefile.txt");
FileEntity entity = new FileEntity(file, "text/plain; charset=\"
UTF-8\"");
HttpPost httppost = new HttpPost("http://localhost/action.do");
httppost.setEntity(entity);
```

请注意InputStreamEntity是不可重复的，因为它仅仅能从低层数据流中读取一次内容。通常来说，我们推荐实现一个定制的HttpEntity类，这是自我包含式的，用来代替使用通用的InputStreamEntity。FileEntity也是一个很好的起点。

1.1.7.1 动态内容实体

通常来说，HTTP实体需要基于特定的执行上下文来动态地生成。通过使用EntityTemplate实体类和ContentProducer接口，HttpClient提供了动态实体的支持。内容生成器是按照需求生成它们内容的对象，将它们写入到一个输出流中。它们是每次被请求时来生成内容。所以用EntityTemplate创建的实体通常是自我包含而且可以重复的。

```
ContentProducer cp = new ContentProducer() {
    public void writeTo(OutputStream outstream) throws IOExcepti
on {
        Writer writer = new OutputStreamWriter(outstream, "UTF-8
");
        writer.write("<response>");
        writer.write(" <content>");
        writer.write(" important stuff");
        writer.write(" </content>");
        writer.write("</response>");
        writer.flush();
    }
};
HttpEntity entity = new EntityTemplate(cp);
HttpPost httppost = new HttpPost("http://localhost/handler.do");
httppost.setEntity(entity);
```

1.1.7.2 HTML表单

许多应用程序需要频繁模拟提交一个HTML表单的过程，比如，为了来记录一个Web应用程序或提交输出数据。HttpClient提供了特殊的实体类UrlEncodedFormEntity来这个满足过程。

```
List<NameValuePair> formparams = new ArrayList<NameValuePair>();
formparams.add(new BasicNameValuePair("param1", "value1"));
formparams.add(new BasicNameValuePair("param2", "value2"));
UrlEncodedFormEntity entity = new UrlEncodedFormEntity(formparam
s, "UTF-8");
HttpPost httppost = new HttpPost("http://localhost/handler.do");
httppost.setEntity(entity);
```

UrlEncodedFormEntity实例将会使用URL编码来编码参数，生成如下的内容：

```
param1=value1&param2=value2
```

1.1.7.3 内容分块

通常，我们推荐让HttpClient选择基于被传递的HTTP报文属性的最适合的编码转换。这是可能的，但是，设置HttpEntity#setChunked()方法为true是通知HttpClient分块编码的首选。请注意HttpClient将会使用标识作为提示。当使用的HTTP协议版本，如HTTP/1.0版本，不支持分块编码时，这个值会被忽略。

```
StringEntity entity = new StringEntity("important message",
    "text/plain; charset=\"UTF-8\"");
entity.setChunked(true);
HttpPost httppost = new HttpPost("http://localhost/acrtion.do");
httppost.setEntity(entity);
```

1.1.8 响应控制器

控制响应的最简便和最方便的方式是使用ResponseHandler接口。这个放完完全减轻了用户关于连接管理的担心。当使用ResponseHandler时，HttpClient将会自动关注并保证释放连接到连接管理器中去，而不管请求执行是否成功或引发了异常。

```
HttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://localhost/");
ResponseHandler<byte[]> handler = new ResponseHandler<byte[]>()
{
    public byte[] handleResponse(
    HttpResponse response) throws ClientProtocolException, IOExc
eption {
        HttpEntity entity = response.getEntity();
        if (entity != null) {
            return EntityUtils.toByteArray(entity);
        } else {
            return null;
        }
    }
};
byte[] response = httpclient.execute(httpget, handler);
```

1.2 HTTP执行的环境

最初，HTTP是被设计成无状态的，面向请求-响应的协议。然而，真实的应用程序经常需要通过一些逻辑相关的请求-响应交换来持久状态信息。为了开启应用程序来维持一个过程状态，HttpClient允许HTTP请求在一个特定的执行环境中来执行，简称为HTTP上下文。如果相同的环境在连续请求之间重用，那么多种逻辑相关的请求可以参与到一个逻辑会话中。HTTP上下文功能和java.util.Map<String,Object>很相似。它仅仅是任意命名参数值的集合。应用程序可以在请求之前或在检查上下文执行完成之后来填充上下文属性。

在HTTP请求执行的这一过程中，HttpClient添加了下列属性到执行上下文中：

- `http.connection` ：HttpConnection实例代表了连接到目标服务器的真实连接。
- `http.target_host` ：HttpHost实例代表了连接目标。
- `http.proxy_host` ：如果使用了，HttpHost实例代表了代理连接。
- `http.request` ：HttpRequest实例代表了真实的HTTP请求。

- `http.response`：HttpResponse实例代表了真实的HTTP响应。
- `http.request_sent`：java.lang.Boolean对象代表了暗示真实请求是否被完全传送到目标连接的标识。

比如，为了决定最终的重定向目标，在请求执行之后，可以检查http.target_host属性的值：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://www.google.com/");
HttpResponse response = httpclient.execute(httpget, localContext
);
HttpHost target = (HttpHost) localContext.getAttribute(
    ExecutionContext.HTTP_TARGET_HOST);
System.out.println("Final target: " + target);
HttpEntity entity = response.getEntity();
if (entity != null) {
    entity.consumeContent();
}
```

输出内容为：

```
Final target: http://www.google.ch
```

1.3 异常处理

HttpClient能够抛出两种类型的异常：在I/O失败时，如套接字连接超时或被重置的java.io.IOException异常，还有标志HTTP请求失败的信号，如违反HTTP协议的HttpException异常。通常I/O错误被认为是非致命的和可以恢复的，而HTTP协议错误则被认为是致命的而且是不能自动恢复的。

1.3.1 HTTP运输安全

要理解HTTP协议并不是对所有类型的应用程序都适合的，这一点很重要。HTTP是一个简单的面向请求/响应的协议，最初被设计用来支持取回静态或动态生成的内容。它从未向支持事务性操作方向发展。比如，如果成功收到和处理请求，HTTP服务器将会考虑它的其中一部分是否完成，生成一个响应并发送一个状态码到客户端。如果客户端因为读取超时，请求取消或系统崩溃导致接收响应实体失败时，服务器不会试图回滚事务。如果客户端决定重新这个请求，那么服务器将不可避免地不止一次执行这个相同的事务。在一些情况下，这会导致应用数据损坏或者不一致的应用程序状态。

尽管HTTP从来都没有被设计来支持事务性处理，但它也能被用作于一个传输协议对关键的任务应用提供被满足的确定状态。要保证HTTP传输层的安全，系统必须保证HTTP方法在应用层的幂等性。

1.3.2 幂等的方法

HTTP/1.1 明确地定义了幂等的方法，描述如下

> 方法也可以有“幂等”属性在那些（除了错误或过期问题）N的副作用>0的相同请求和独立的请求是相同的

换句话说，应用程序应该保证准备着来处理多个相同方法执行的实现。这是可以达到的，比如，通过提供一个独立的事务ID和其它避免执行相同逻辑操作的方法。

请注意这个问题对于HttpClient是不具体的。基于应用的浏览器特别受和非幂等的HTTP方法相关的相同问题的限制。

HttpClient假设没有实体包含方法，比如GET和HEAD是幂等的，而实体包含方法，比如POST和PUT则不是。

1.3.3 异常自动恢复

默认情况下，HttpClient会试图自动从I/O异常中恢复。默认的自动恢复机制是受很少一部分已知的异常是安全的这个限制。

- HttpClient不会从任意逻辑或HTTP协议错误（那些是从HttpException类中派生出的）中恢复的。
- HttpClient将会自动重新执行那么假设是幂等的方法。
- HttpClient将会自动重新执行那些由于运输异常失败，而HTTP请求仍然被传送到目标服务器（也就是请求没有完全被送到服务器）失败的方法。
- HttpClient将会自动重新执行那些已经完全被送到服务器，但是服务器使用HTTP状态码（服务器仅仅丢掉连接而不会发回任何东西）响应时失败的方法。在这种情况下，假设请求没有被服务器处理，而应用程序的状态也没有改变。如果这个假设可能对于你应用程序的目标Web服务器来说不正确，那么就强烈建议提供一个自定义的异常处理器。

1.3.4 请求重试处理

为了开启自定义异常恢复机制，应该提供一个HttpRequestRetryHandler接口的实现。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpRequestRetryHandler myRetryHandler = new HttpRequestRetryHan
dler() {
    public boolean retryRequest(IOException exception,
    int executionCount,HttpContext context) {
        if (executionCount >= 5) {
            // 如果超过最大重试次数，那么就不要继续了
            return false;
        }
        if (exception instanceof NoHttpResponseException) {
            // 如果服务器丢掉了连接，那么就重试
            return true;
        }
        if (exception instanceof SSLHandshakeException) {
            // 不要重试SSL握手异常
            return false;
        }
        HttpRequest request = (HttpRequest) context.getAttribute
(
            ExecutionContext.HTTP_REQUEST);
        boolean idempotent = !(request instanceof HttpEntityEncl
osingRequest);
        if (idempotent) {
            // 如果请求被认为是幂等的，那么就重试
            return true;
        }
        return false;
    }
};
httpclient.setHttpRequestRetryHandler(myRetryHandler);
```

1.4 中止请求

在一些情况下，由于目标服务器的高负载或客户端有很多活动的请求，那么HTTP请求执行会在预期的时间框内而失败。这时，就可能不得不过早地中止请求，解除封锁在I/O执行中的线程封锁。被HttpClient执行的HTTP请求可以在执行的任意阶段通过调用HttpUriRequest#abort()方法而中止。这个方法是线程安全的，而且可以从任意线程中调用。当一个HTTP请求被中止时，它的执行线程就封锁在I/O操作中了，而且保证通过抛出InterruptedIOException异常来解锁。

1.5 HTTP协议拦截器

HTTP协议拦截器是一个实现了特定HTPP协议方面的惯例。通常协议拦截器希望作用于一个特定头部信息上，或者一族收到报文的相关头部信息，或使用一个特定的头部或一族相关的头部信息填充发出的报文。协议拦截器也可以操纵包含在报文中

的内容实体，透明的内容压缩/解压就是一个很好的示例。通常情况下这是由包装器实体类使用了“装饰者”模式来装饰原始的实体完成的。一些协议拦截器可以从一个逻辑单元中来结合。

协议拦截器也可以通过共享信息来共同合作-比如处理状态-通过HTTP执行上下文。协议拦截器可以使用HTTP内容来为一个或多个连续的请求存储一个处理状态。

通常拦截器执行的顺序不应该和它们基于的特定执行上下文状态有关。如果协议拦截器有相互依存关系，那么它们必须按特定顺序来执行，正如它们希望执行的顺序一样，它们应该在相同的序列中被加到协议处理器。

协议拦截器必须实现为线程安全的。和Servlet相似，协议拦截器不应该使用实例变量，除非访问的那些变量是同步的。

这个示例给出了本地内容在连续的请求中怎么被用于持久一个处理状态的：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
AtomicInteger count = new AtomicInteger(1);
localContext.setAttribute("count", count);
httpclient.addRequestInterceptor(new HttpRequestInterceptor() {
    public void process(final HttpRequest request,
    final HttpContext context) throws HttpException, IOException
 {
        AtomicInteger count = (AtomicInteger) context.getAttribu
te("count");
        request.addHeader("Count", Integer.toString(count.getAnd
Increment()));
    }
});
HttpGet httpget = new HttpGet("http://localhost/");
for (int i = 0; i < 10; i++) {
    HttpResponse response = httpclient.execute(httpget, localCon
text);
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        entity.consumeContent();
    }
}
```

1.6 HTTP参数

HttpParams接口代表了定义组件运行时行为的一个不变的值的集合。很多情况下，HttpParams和HttpContext相似。二者之间的主要区别是它们在运行时使用的不同。这两个接口表示了对象的集合，它们被视作为访问对象值的键的Map，但是服务于不同的目的：

- HttpParams旨在包含简单对象：整型，浮点型，字符串，集合，还有运行时不变的对象。

- HttpParams希望被用在“一次写入-多处准备”模式下。HttpContext旨在包含很可能在HTTP报文处理这一过程中发生改变的复杂对象
- HttpParams的目标是定义其它组件的行为。通常每一个复杂的组件都有它自己的HttpParams对象。HttpContext的目标是来表示一个HTTP处理的执行状态。通常相同的执行上下文在很多合作的对象中共享。

1.6.1 参数层次

在HTTP请求执行过程中，HttpRequest对象的HttpParams是和用于执行请求的HttpClient实例的HttpParams联系在一起的。这使得设置在HTTP请求级别的参数优先于设置在HTTP客户端级别的HttpParams。推荐的做法是设置普通参数对所有的在HTTP客户端级别的HTTP请求共享，而且可以选择性重写具体在HTTP请求级别的参数。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.getParams().setParameter(CoreProtocolPNames.PROTOCOL_
VERSION,HttpVersion.HTTP_1_0);
httpclient.getParams().setParameter(CoreProtocolPNames.HTTP_CONT
ENT_CHARSET,"UTF-8");
HttpGet httpget = new HttpGet("http://www.google.com/");
httpget.getParams().setParameter(CoreProtocolPNames.PROTOCOL_VER
SION,HttpVersion.HTTP_1_1);
httpget.getParams().setParameter(CoreProtocolPNames.USE_EXPECT_C
ONTINUE,Boolean.FALSE);
httpclient.addRequestInterceptor(new HttpRequestInterceptor() {
    public void process(final HttpRequest request,
        final HttpContext context) throws HttpException, IOExcep
tion {
        System.out.println(request.getParams().getParameter(
            CoreProtocolPNames.PROTOCOL_VERSION));
        System.out.println(request.getParams().getParameter(
            CoreProtocolPNames.HTTP_CONTENT_CHARSET));
        System.out.println(request.getParams().getParameter(
            CoreProtocolPNames.USE_EXPECT_CONTINUE));
        System.out.println(request.getParams().getParameter(
            CoreProtocolPNames.STRICT_TRANSFER_ENCODING));
    }
});
```

输出内容为：

```
HTTP/1.1
UTF-8
false
null
```

1.6.2 HTTP参数bean

HttpParams接口允许在处理组件的配置上很大的灵活性。很重要的是，新的参数可以被引入而不会影响老版本的二进制兼容性。然而，和常规的Java bean相比，HttpParams也有一个缺点：HttpParams不能使用DI框架来组合。为了缓解这个限制，HttpClient包含了一些bean类，它们可以用来按顺序使用标准的Java eban惯例初始化HttpParams对象。

```
HttpParams params = new BasicHttpParams();
HttpProtocolParamBean paramsBean = new HttpProtocolParamBean(par
ams);
paramsBean.setVersion(HttpVersion.HTTP_1_1);
paramsBean.setContentCharset("UTF-8");
paramsBean.setUseExpectContinue(true);
System.out.println(params.getParameter(
    CoreProtocolPNames.PROTOCOL_VERSION));
System.out.println(params.getParameter(
    CoreProtocolPNames.HTTP_CONTENT_CHARSET));
System.out.println(params.getParameter(
    CoreProtocolPNames.USE_EXPECT_CONTINUE));
System.out.println(params.getParameter(
    CoreProtocolPNames.USER_AGENT));
```

输出内容为：

```
HTTP/1.1
UTF-8
false
null
```

1.7 HTTP请求执行参数

这些参数会影响到请求执行的过程：

- `http.protocol.version` ：如果没有在请求对象中设置明确的版本信息，它就定义了使用的HTTP协议版本。这个参数期望得到一个ProtocolVersion类型的值。如果这个参数没有被设置，那么就使用HTTP/1.1。
- `http.protocol.element-charset` ：定义了编码HTTP协议元素的字符集。这个参数期望得到一个java.lang.String类型的值。如果这个参数没有被设置，那么就使用US-ASCII。
- `http.protocol.eontent-charset` ：定义了为每个内容主体编码的默认字符集。这个参数期望得到一个java.lang.String类型的值。如果这个参数没有被设置，那么就使用ISO-8859-1。
- `http.useragent` ：定义了头部信息User-Agent的内容。这个参数期望得到一个java.lang.String类型的值。如果这个参数没有被设置，那么HttpClient将会为它自动生成一个值。
- `http.protocol.strict-transfer-encoding` ：定义了响应头部信息中是否含有一个非法的Transfer-Encoding，都要拒绝掉。

- `http.protocol.expect-continue` ：为包含方法的实体激活Expect: 100-Continue握手。Expect: 100-Continue握手的目的是允许客户端使用请求体发送一个请求信息来决定源服务器是否希望在客户端发送请求体之前得到这个请求（基于请求头部信息）。Expect: 100-Continue握手的使用可以对需要目标服务器认证的包含请求的实体（比如POST和PUT）导致明显的性能改善。Expect: 100-Continue握手应该谨慎使用，因为它和HTTP服务器，不支持HTTP/1.1协议的代理使用会引起问题。这个参数期望得到一个java.lang.Boolean类型的值。如果这个参数没有被设置，那么HttpClient将会试图使用握手。
- `http.protocol.wait-for-continue` ：定义了客户端应该等待100-Continue响应最大的毫秒级时间间隔。这个参数期望得到一个java.lang.Integer类型的值。如果这个参数没有被设置，那么HttpClient将会在恢复请求体传输之前为确认等待3秒。

第二章 连接管理

HttpClient有一个对连接初始化和终止，还有在活动连接上I/O操作的完整控制。而连接操作的很多方面可以使用一些参数来控制。

2.1 连接参数

这些参数可以影响连接操作：

- `http.socket.timeout`：定义了套接字的毫秒级超时时间（SO_TIMEOUT），这就是等待数据，换句话说，在两个连续的数据包之间最大的闲置时间。如果超时时间是0就解释为是一个无限大的超时时间。这个参数期望得到一个java.lang.Integer类型的值。如果这个参数没有被设置，那么读取操作就不会超时（无限大的超时时间）。
- `http.tcp.nodelay`：决定了是否使用Nagle算法。Nagle算法视图通过最小化发送的分组数量来节省带宽。当应用程序希望降低网络延迟并提高性能时，它们可以关闭Nagle算法（也就是开启TCP_NODELAY）。数据将会更早发送，增加了带宽消耗的成文。这个参数期望得到一个java.lang.Boolean类型的值。如果这个参数没有被设置，那么TCP_NODELAY就会开启（无延迟）。
- `http.socket.buffer-size`：决定了内部套接字缓冲使用的大小，来缓冲数据同时接收/传输HTTP报文。这个参数期望得到一个java.lang.Integer类型的值。如果这个参数没有被设置，那么HttpClient将会分配8192字节的套接字缓存。
- `http.socket.linger`：使用指定的秒数拖延时间来设置SO_LINGER。最大的连接超时值是平台指定的。值0暗示了这个选项是关闭的。值-1暗示了使用了JRE默认的。这个设置仅仅影响套接字关闭操作。如果这个参数没有被设置，那么就假设值为-1（JRE默认）。
- `http.connection.timeout`：决定了直到连接建立时的毫秒级超时时间。超时时间的值为0解释为一个无限大的时间。这个参数期望得到一个java.lang.Integer类型的值。如果这个参数没有被设置，连接操作将不会超时（无限大的超时时间）。
- `http.connection.stalecheck`：决定了是否使用旧的连接检查。当在一个连接之上执行一个请求而服务器端的连接已经关闭时，关闭旧的连接检查可能导致在获得一个I/O错误风险时显著的性能提升（对于每一个请求，检查时间可以达到30毫秒）。这个参数期望得到一个java.lang.Boolean类型的值。出于性能的关键操作，检查应该被关闭。如果这个参数没有被设置，那么旧的连接将会在每个请求执行之前执行。
- `http.connection.max-line-length`：决定了最大请求行长度的限制。如果设置为一个正数，任何HTTP请求行超过这个限制将会引发java.io.IOException异常。负数或零将会关闭这个检查。这个参数期望得到一个java.lang.Integer类型的值。如果这个参数没有被设置，那么就不强制进行限制了。
- `http.connection.max-header-count`：决定了允许的最大HTTP头部信息数量。如果设置为一个正数，从数据流中获得的HTTP头部信息数量超过这个

限制就会引发java.io.IOException异常。负数或零将会关闭这个检查。这个参数期望得到一个java.lang.Integer类型的值。如果这个参数没有被设置，那么就不
- 强制进行限制了。
- `http.connection.max-status-line-garbage`：决定了在期望得到HTTP响应状态行之前可忽略请求行的最大数量。使用HTTP/1.1持久性连接，这个问题产生的破碎的脚本将会返回一个错误的Content-Length（有比指定的字节更多的发送）。不幸的是，在某些情况下，这个不能在错误响应后来侦测，只能在下一次之前。所以HttpClient必须以这种方式跳过那些多余的行。这个参数期望得到一个java.lang.Integer类型的值。0是不允许在状态行之前的所有垃圾/空行。使用java.lang.Integer#MAX_VALUE来设置不限制的数字。如果这个参数没有被设置那就假设是不限制的。

2.2 持久连接

从一个主机向另外一个建立连接的过程是相当复杂的，而且包含了两个终端之间的很多包的交换，它是相当费时的。连接握手的开销是很重要的，特别是对小量的HTTP报文。如果打开的连接可以被重用来执行多次请求，那么就可以达到很高的数据吞吐量。

HTTP/1.1强调HTTP连接默认情况可以被重用于多次请求。HTTP/1.0兼容的终端也可以使用相似的机制来明确地交流它们的偏好来保证连接处于活动状态，也使用它来处理多个请求。HTTP代理也可以保持空闲连接处于一段时间的活动状态，防止对相同目标主机的一个连接也许对随后的请求需要。保持连接活动的能力通常被称作持久性连接。HttpClient完全支持持久性连接。

2.3 HTTP连接路由

HttpClient能够直接或通过路由建立连接到目标主机，这会涉及多个中间连接，也被称为跳。HttpClient区分路由和普通连接，通道和分层。通道连接到目标主机的多个中间代理的使用也称作是代理链。

普通路由由连接到目标或仅第一次的代理来创建。通道路由通过代理链到目标连接到第一通道来建立。没有代理的路由不是通道的，分层路由通过已存在连接的分层协议来建立。协议仅仅可以在到目标的通道上或在没有代理的直接连接上分层。

2.3.1 路由计算

RouteInfo接口代表关于最终涉及一个或多个中间步骤或跳的目标主机路由的信息。HttpRoute是RouteInfo的具体实现，这是不能改变的（是不变的）。HttpTracker是可变的RouteInfo实现，由HttpClient在内部使用来跟踪到最大路由目标的剩余跳数。HttpTracker可以在成功执行向路由目标的下一跳之后更新。HttpRouteDirector是一个帮助类，可以用来计算路由中的下一跳。这个类由HttpClient在内部使用。

HttpRoutePlanner是一个代表计算到基于执行上下文到给定目标完整路由策略的接口。HttpClient附带两个默认的HttpRoutePlanner实现。ProxySelectorRoutePlanner是基于java.net.ProxySelector的。默认情况下，它会从系统属性中或从运行应用程序的浏览器中选取JVM的代理设置。DefaultHttpRoutePlanner实现既不使用任何Java系统属性，也不使用系统或浏览器的代理设置。它只基于HTTP如下面描述的参数计算路由。

2.3.2 安全HTTP连接

如果信息在两个不能由非认证的第三方进行读取或修改的终端之间传输，HTTP连接可以被认为是安全的。SSL/TLS协议是用来保证HTTP传输安全使用最广泛的技术。而其它加密技术也可以被使用。通常来说，HTTP传输是在SSL/TLS加密连接之上分层的。

2.4 HTTP路由参数

这些参数可以影响路由计算：

- `http.route.default-proxy`：定义可以被不使用JRE设置的默认路由规划者使用的代理主机。这个参数期望得到一个HttpHost类型的值。如果这个参数没有被设置，那么就会尝试直接连接到目标。
- `http.route.local-address`：定义一个本地地址由所有默认路由规划者来使用。有多个网络接口的机器中，这个参数可以被用于从连接源中选择网络接口。这个参数期望得到一个java.net.InetAddress类型的值。如果这个参数没有被设置，将会自动使用本地地址。
- `http.route.forced-route`：定义一个由所有默认路由规划者使用的强制路由。代替了计算路由，给定的强制路由将会被返回，尽管它指向一个完全不同的目标主机。这个参数期望得到一个HttpRoute类型的值。如果这个参数没有被设置，那么就使用默认的规则建立连接到目标服务器。

2.5 套接字工厂

LayeredSocketFactory是SocketFactory接口的扩展。分层的套接字工厂可HTTP连接内部使用java.net.Socket对象来处理数据在线路上的传输。它们依赖SocketFactory接口来创建，初始化和连接套接字。这会使得HttpClient的用户可以提供在运行时指定套接字初始化代码的应用程序。PlainSocketFactory是创建和初始化普通的（不加密的）套接字的默认工厂。

创建套接字的过程和连接到主机的过程是不成对的，所以套接字在连接操作封锁时可以被关闭。

```
PlainSocketFactory sf = PlainSocketFactory.getSocketFactory();
Socket socket = sf.createSocket();
HttpParams params = new BasicHttpParams();
params.setParameter(CoreConnectionPNames.CONNECTION_TIMEOUT, 100
0L);
sf.connectSocket(socket, "locahost", 8080, null, -1, params);
```

2.5.1 安全套接字分层

LayeredSocketFactory是SocketFactory接口的扩展。分层的套接字工厂可以创建在已经存在的普通套接字之上的分层套接字。套接字分层主要通过代理来创建安全的套接字。HttpClient附带实现了SSL/TLS分层的SSLSocketFactory。请注意HttpClient不使用任何自定义加密功能。它完全依赖于标准的Java密码学（JCE）和安全套接字（JSEE）扩展。

2.5.2 SSL/TLS的定制

HttpClient使用SSLSocketFactory来创建SSL连接。SSLSocketFactory允许高度定制。它可以使用javax.net.ssl.SSLContext的实例作为参数，并使用它来创建定制SSL连接。

```
TrustManager easyTrustManager = new X509TrustManager() {
    @Override
    public void checkClientTrusted(X509Certificate[] chain,
        String authType) throws CertificateException {
        // 哦，这很简单！
    }
    @Override
    public void checkServerTrusted(X509Certificate[] chain,
        String authType) throws CertificateException {
        //哦，这很简单！
    }
    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
};
SSLContext sslcontext = SSLContext.getInstance("TLS");
sslcontext.init(null, new TrustManager[] { easyTrustManager }, n
ull);
SSLSocketFactory sf = new SSLSocketFactory(sslcontext);
SSLSocket socket = (SSLSocket) sf.createSocket();
socket.setEnabledCipherSuites(new String[] { "SSL_RSA_WITH_RC4_1
28_MD5" });
HttpParams params = new BasicHttpParams();
params.setParameter(CoreConnectionPNames.CONNECTION_TIMEOUT, 100
0L);
sf.connectSocket(socket, "locahost", 443, null, -1, params);
```

SSLSocketFactory的定制暗示出一定程度SSL/TLS协议概念的熟悉，这个详细的解释超出了本文档的范围。请参考Java的安全套接字扩展，这是javax.net.ssl.SSLContext和相关工具的详细描述。

2.5.3 主机名验证

除了信任验证和客户端认证在SSL/TLS协议级上进行，一旦连接建立之后，HttpClient能可选地验证目标主机名匹配存储在服务器的X.509认证中的名字。这个认证可以提供额外的服务器信任材料的真实保证。X509主机名验证接口代表了主机名验证的策略。HttpClient附带了3个X509主机名验证器。很重要的一点是：主机名验证不应该混淆SSL信任验证。

- StrictHostnameVerifier：严格的主机名验证在Sun Java 1.4，Sun Java 5和Sun Java 6中是相同的。而且也非常接近IE6。这个实现似乎是兼容RFC 2818处理通配符的。主机名必须匹配第一个CN或任意的subject-alt。在CN和其它任意的subject-alt中可能会出现通配符。
- BrowserCompatHostnameVerifier：主机名验证器和Curl和Firefox的工作方式是相同的。主机名必须匹配第一个CN或任意的subject-alt。在CN和其它任意的subject-alt中可能会出现通配符。BrowserCompatHostnameVerifier和StrictHostnameVerifier的唯一不同是使用BrowserCompatHostnameVerifier匹配所有子域的通配符（比如"*.foo.com"），包括"a.b.foo.com"。

- AllowAllHostnameVerifier：这个主机名验证器基本上是关闭主机名验证的。这个实现是一个空操作，而且不会抛出javax.net.ssl.SSLException异常。

每一个默认的HttpClient使用BrowserCompatHostnameVerifier的实现。如果需要的话，它可以指定不同的主机名验证器实现。

```
SSLSocketFactory sf = new SSLSocketFactory(SSLContext.getInstanc
e("TLS"));
sf.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERIFIER
);
```

2.6 协议模式

Scheme类代表了一个协议模式，比如“http”或“https”同时包含一些协议属性，比如默认端口，用来为给定协议创建java.net.Socket实例的套接字工厂。SchemeRegistry类用来维持一组Scheme，当去通过请求URI建立连接时，HttpClient可以从中选择：

```
Scheme http = new Scheme("http", PlainSocketFactory.getSocketFac
tory(), 80);
SSLSocketFactory sf = new SSLSocketFactory(SSLContext.getInstanc
e("TLS"));
sf.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERIFIER
);
Scheme https = new Scheme("https", sf, 443);
SchemeRegistry sr = new SchemeRegistry();
sr.register(http);
sr.register(https);
```

2.7 HttpClient代理配置

尽管HttpClient了解复杂的路由模式和代理链，它仅支持简单直接的或开箱的跳式代理连接。

告诉HttpClient通过代理去连接到目标主机的最简单方式是通过设置默认的代理参数：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpHost proxy = new HttpHost("someproxy", 8080);
httpclient.getParams().setParameter(ConnRoutePNames.DEFAULT_PROX
Y, proxy);
```

也可以构建HttpClient使用标准的JRE代理选择器来获得代理信息：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
ProxySelectorRoutePlanner routePlanner = new ProxySelectorRouteP
lanner(
    httpclient.getConnectionManager().getSchemeRegistry(),
    ProxySelector.getDefault());
httpclient.setRoutePlanner(routePlanner);
```

另外一种选择，可以提供一个定制的RoutePlanner实现来获得HTTP路由计算处理上的复杂的控制：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setRoutePlanner(new HttpRoutePlanner() {
    public HttpRoute determineRoute(HttpHost target,
        HttpRequest request,
        HttpContext context) throws HttpException {
        return new HttpRoute(target, null, new HttpHost("somepro
xy", 8080),
            "https".equalsIgnoreCase(target.getSchemeName()));
    }
});
```

2.8 HTTP连接管理器

2.8.1 连接操作器

连接操作是客户端的低层套接字或可以通过外部实体，通常称为连接操作的被操作的状态的连接。OperatedClientConnection接口扩展了HttpClientConnection接口而且定义了额外的控制连接套接字的方法。ClientConnectionOperator接口代表了创建实例和更新那些对象低层套接字的策略。实现类最有可能利用SocketFactory来创建java.net.Socket实例。ClientConnectionOperator接口可以让HttpClient的用户提供一个连接操作的定制策略和提供可选实现OperatedClientConnection接口的能力。

2.8.2 管理连接和连接管理器

HTTP连接是复杂的，有状态的，线程不安全的对象需要正确的管理以便正确地执行功能。HTTP连接在同一时间仅仅只能由一个执行线程来使用。HttpClient采用一个特殊实体来管理访问HTTP连接，这被称为HTTP连接管理器，代表了ClientConnectionManager接口。一个HTTP连接管理器的目的是作为工厂服务于新的HTTP连接，管理持久连接和同步访问持久连接来确保同一时间仅有一个线程可以访问一个连接。

内部的HTTP连接管理器和OperatedClientConnection实例一起工作，但是它们为服务消耗器ManagedClientConnection提供实例。ManagedClientConnection扮演连接之上管理状态控制所有I/O操作的OperatedClientConnection实例的包装器。它也

抽象套接字操作，提供打开和更新去创建路由套接字便利的方法。ManagedClientConnection实例了解产生它们到连接管理器的链接，而且基于这个事实，当不再被使用时，它们必须返回到管理器。ManagedClientConnection类也实现了ConnectionReleaseTrigger接口，可以被用来触发释放连接返回给管理器。一旦释放连接操作被触发了，被包装的连接从ManagedClientConnection包装器中脱离，OperatedClientConnection实例被返回给管理器。尽管服务消耗器仍然持有ManagedClientConnection实例的引用，它也不再去执行任何I/O操作或有意无意地改变的OperatedClientConnection状态。

这里有一个从连接管理器中获取连接的示例：

```
HttpParams params = new BasicHttpParams();
Scheme http = new Scheme("http", PlainSocketFactory.getSocketFac
tory(), 80);
SchemeRegistry sr = new SchemeRegistry();
sr.register(http);
ClientConnectionManager connMrg = new SingleClientConnManager(pa
rams, sr);
// 请求新连接。这可能是一个很长的过程。
ClientConnectionRequest connRequest = connMrg.requestConnection(
    new HttpRoute(new HttpHost("localhost", 80)), null);
// 等待连接10秒
ManagedClientConnection conn = connRequest.getConnection(10, Tim
eUnit.SECONDS);
try {
    // 用连接在做有用的事情。当完成时释放连接。
    conn.releaseConnection();
} catch (IOException ex) {
    // 在I/O error之上终止连接。
    conn.abortConnection();
    throw ex;
}
```

如果需要，连接请求可以通过调用来ClientConnectionRequest#abortRequest()方法过早地中断。这会解锁在ClientConnectionRequest#getConnection()方法中被阻止的线程。

一旦响应内容被完全消耗后，BasicManagedEntity包装器类可以用来保证自动释放低层的连接。HttpClient内部使用这个机制来实现透明地对所有从HttpClient#execute()方法中获得响应释放连接：

```
ClientConnectionRequest connRequest = connMrg.requestConnection(
    new HttpRoute(new HttpHost("localhost", 80)), null);
ManagedClientConnection conn = connRequest.getConnection(10, Tim
eUnit.SECONDS);
try {
    BasicHttpRequest request = new BasicHttpRequest("GET", "/");
    conn.sendRequestHeader(request);
    HttpResponse response = conn.receiveResponseHeader();
    conn.receiveResponseEntity(response);
    HttpEntity entity = response.getEntity();
    if (entity != null) {
        BasicManagedEntity managedEntity = new BasicManagedEntit
y(entity, conn, true);
        // 替换实体
        response.setEntity(managedEntity);
    }
    // 使用响应对象做有用的事情。当响应内容被消耗后这个连接将会自动释放。
} catch (IOException ex) {
    //在I/O error之上终止连接。
    conn.abortConnection();
    throw ex;
}
```

2.8.3 简单连接管理器

SingleClientConnManager是一个简单的连接管理器，在同一时间它仅仅维护一个连接。尽管这个类是线程安全的，但它应该被用于一个执行线程。SingleClientConnManager对于同一路由的后续请求会尽量重用连接。而如果持久连接的路由不匹配连接请求的话，它也会关闭存在的连接之后对给定路由再打开一个新的。如果连接已经被分配，将会抛出java.lang.IllegalStateException异常。

对于每个默认连接，HttpClient使用SingleClientConnManager。

2.8.4 连接池管理器

ThreadSafeClientConnManager是一个复杂的实现来管理客户端连接池，它也可以从多个执行线程中服务连接请求。对每个基本的路由，连接都是池管理的。对于路由的请求，管理器在池中有可用的持久性连接，将被从池中租赁连接服务，而不是创建一个新的连接。

ThreadSafeClientConnManager维护每个基本路由的最大连接限制。每个默认的实现对每个给定路由将会创建不超过两个的并发连接，而总共也不会超过20个连接。对于很多真实的应用程序，这个限制也证明很大的制约，特别是他们在服务中使用HTTP作为传输协议。连接限制，也可以使用HTTP参数来进行调整。

这个示例展示了连接池参数是如何来调整的：

```
HttpParams params = new BasicHttpParams();
// 增加最大连接到200
ConnManagerParams.setMaxTotalConnections(params, 200);
// 增加每个路由的默认最大连接到20
ConnPerRouteBean connPerRoute = new ConnPerRouteBean(20);
// 对localhost:80增加最大连接到50
HttpHost localhost = new HttpHost("locahost", 80);
connPerRoute.setMaxForRoute(new HttpRoute(localhost), 50);
ConnManagerParams.setMaxConnectionsPerRoute(params, connPerRoute
);
SchemeRegistry schemeRegistry = new SchemeRegistry();
schemeRegistry.register(
    new Scheme("http", PlainSocketFactory.getSocketFactory(), 80
));
schemeRegistry.register(
    new Scheme("https", SSLSocketFactory.getSocketFactory(), 443
));
ClientConnectionManager cm = new ThreadSafeClientConnManager(par
ams, schemeRegistry);
HttpClient httpClient = new DefaultHttpClient(cm, params);
```

2.8.5 连接管理器关闭

当一个HttpClient实例不再需要时，而且即将走出使用范围，那么关闭连接管理器来保证由管理器保持活动的所有连接被关闭，由连接分配的系统资源被释放是很重要的。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpGet httpget = new HttpGet("http://www.google.com/");
HttpResponse response = httpclient.execute(httpget);
HttpEntity entity = response.getEntity();
System.out.println(response.getStatusLine());
if (entity != null) {
    entity.consumeContent();
}
httpclient.getConnectionManager().shutdown();
```

2.9 连接管理参数

这些是可以用于定制标准HTTP连接管理器实现的参数：

- `http.conn-manager.timeout`：定义了当从ClientConnectionManager中检索ManagedClientConnection实例时使用的毫秒级的超时时间。这个参数期望得到一个java.lang.Long类型的值。如果这个参数没有被设置，连接请求就不会超时（无限大的超时时间）。
- `http.conn-manager.max-per-route`：定义了每个路由连接的最大数量。这个限制由客户端连接管理器来解释，而且应用于独立的管理器实例。这个参

数期望得到一个ConnPerRoute类型的值。
- `http.conn-manager.max-total`：定义了总共连接的最大数目。这个限制由客户端连接管理器来解释，而且应用于独立的管理器实例。这个参数期望得到一个java.lang.Integer类型的值。

2.10 多线程执行请求

当配备连接池管理器时，比如ThreadSafeClientConnManager，HttpClient可以同时被用来执行多个请求，使用多线程执行。

ThreadSafeClientConnManager将会分配基于它的配置的连接。如果对于给定路由的所有连接都被租出了，那么连接的请求将会阻塞，直到一个连接被释放回连接池。它可以通过设置'http.conn-manager.timeout'为一个正数来保证连接管理器不会在连接请求执行时无限期的被阻塞。如果连接请求不能在给定的时间周期内被响应，将会抛出ConnectionPoolTimeoutException异常。

```
HttpParams params = new BasicHttpParams();
SchemeRegistry schemeRegistry = new SchemeRegistry();
schemeRegistry.register(
    new Scheme("http", PlainSocketFactory.getSocketFactory(), 80
));
ClientConnectionManager cm = new ThreadSafeClientConnManager(par
ams, schemeRegistry);
HttpClient httpClient = new DefaultHttpClient(cm, params);
// 执行GET方法的URI
String[] urisToGet = {
    "http://www.domain1.com/",
    "http://www.domain2.com/",
    "http://www.domain3.com/",
    "http://www.domain4.com/"
};
// 为每个URI创建一个线程
GetThread[] threads = new GetThread[urisToGet.length];
for (int i = 0; i < threads.length; i++) {
    HttpGet httpget = new HttpGet(urisToGet[i]);
    threads[i] = new GetThread(httpClient, httpget);
}
```

```
// 开始执行线程
for (int j = 0; j < threads.length; j++) {
    threads[j].start();
}
// 合并线程
for (int j = 0; j < threads.length; j++) {
    threads[j].join();
}

static class GetThread extends Thread {
    private final HttpClient httpClient;
    private final HttpContext context;
    private final HttpGet httpget;
    public GetThread(HttpClient httpClient, HttpGet httpget) {
        this.httpClient = httpClient;
        this.context = new BasicHttpContext();
        this.httpget = httpget;
    }
    @Override
    public void run() {
        try {
            HttpResponse response = this.httpClient.execute(this
.httpget, this.context);
            HttpEntity entity = response.getEntity();
            if (entity != null) {
                // 对实体做些有用的事情...
                // 保证连接能释放回管理器
                entity.consumeContent();
            }
        } catch (Exception ex) {
            this.httpget.abort();
        }
    }
}
```

2.11 连接收回策略

一个经典的阻塞I/O模型的主要缺点是网络套接字仅当I/O操作阻塞时才可以响应I/O事件。当一个连接被释放返回管理器时，它可以被保持活动状态而却不能监控套接字的状态和响应任何I/O事件。如果连接在服务器端关闭，那么客户端连接也不能去侦测连接状态中的变化和关闭本端的套接字去作出适当响应。

HttpClient通过测试连接是否是过时的来尝试去减轻这个问题，这已经不再有效了，因为它已经在服务器端关闭了，之前使用执行HTTP请求的连接。过时的连接检查也并不是100%的稳定，反而对每次请求执行还要增加10到30毫秒的开销。唯一可行的而不涉及到每个对空闲连接的套接字模型线程解决方案，是使用专用的监控线程来收回因为长时间不活动而被认为是过期的连接。监控线程可以周期地调用ClientConnectionManager#closeExpiredConnections()方法来关闭所有过期的连

接，从连接池中收回关闭的连接。它也可以选择性调用ClientConnectionManager#closeIdleConnections()方法来关闭所有已经空闲超过给定时间周期的连接。

```
public static class IdleConnectionMonitorThread extends Thread {
    private final ClientConnectionManager connMgr;
    private volatile boolean shutdown;
    public IdleConnectionMonitorThread(ClientConnectionManager c
onnMgr) {
        super();
        this.connMgr = connMgr;
    }
    @Override
    public void run() {
        try {
            while (!shutdown) {
                synchronized (this) {
                    wait(5000);
                    // 关闭过期连接
                    connMgr.closeExpiredConnections();
                    // 可选地，关闭空闲超过30秒的连接
                    connMgr.closeIdleConnections(30, TimeUnit.SE
CONDS);
                }
            }
        } catch (InterruptedException ex) {
            // 终止
        }
    }
    public void shutdown() {
        shutdown = true;
        synchronized (this) {
            notifyAll();
        }
    }
}
```

2.12 连接保持活动的策略

HTTP规范没有确定一个持久连接可能或应该保持活动多长时间。一些HTTP服务器使用非标准的头部信息Keep-Alive来告诉客户端它们想在服务器端保持连接活动的周期秒数。如果这个信息可用，HttClient就会利用这个它。如果头部信息Keep-Alive在响应中不存在，HttpClient假设连接无限期的保持活动。然而许多现实中的HTTP服务器配置了在特定不活动周期之后丢掉持久连接来保存系统资源，往往这是不通知客户端的。如果默认的策略证明是过于乐观的，那么就会有人想提供一个定制的保持活动策略。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setKeepAliveStrategy(new ConnectionKeepAliveStrategy(
) {
    public long getKeepAliveDuration(HttpResponse response, Http
Context context) {
        // 兑现'keep-alive'头部信息
        HeaderElementIterator it = new BasicHeaderElementIterato
r(
            response.headerIterator(HTTP.CONN_KEEP_ALIVE));
        while (it.hasNext()) {
            HeaderElement he = it.nextElement();
            String param = he.getName(); String value = he.getVa
lue();
            if (value != null && param.equalsIgnoreCase("timeout
")) {
                try {
                    return Long.parseLong(value) * 1000;
                } catch(NumberFormatException ignore) {
                }
            }
        }
        HttpHost target = (HttpHost) context.getAttribute(
            ExecutionContext.HTTP_TARGET_HOST);
        if ("www.naughty-server.com".equalsIgnoreCase(target.get
HostName())) {
            // 只保持活动5秒
            return 5 * 1000;
        } else {
            // 否则保持活动30秒
            return 30 * 1000;
        }
    }
});
```

第三章 HTTP状态管理

原始的HTTP是被设计为无状态的，面向请求/响应的协议，没有特殊规定有状态的，贯穿一些逻辑相关的请求/响应交换的会话。由于HTTP协议变得越来越普及和受欢迎，越来越多的从前没有打算使用它的系统也开始为应用程序来使用它，比如作为电子商务应用程序的传输方式。因此，支持状态管理就变得非常必要了。

网景公司，一度成为Web客户端和服务器软件开发者的领导方向，在它们基于专有规范的产品中实现了对HTTP状态管理的支持。之后，网景公司试图通过发布规范草案来规范这种机制。它们的努力通过RFC标准跟踪促成了这些规范定义。然而，在很多应用程序中的状态管理仍然基于网景公司的草案而不兼容官方的规范。很多主要的Web浏览器开发者觉得有必要保留那些极大促进标准片段应用程序的兼容性。

3.1 HTTP cookies

Cookie是HTTP代理和目标服务器可以交流保持会话的状态信息的令牌或短包。网景公司的工程师用它来指“魔法小甜饼”和粘住的名字。

HttpClient使用Cookie接口来代表抽象的cookie令牌。在它的简单形式中HTTP的cookie几乎是名/值对。通常一个HTTP的cookie也包含一些属性，比如版本号，合法的域名，指定cookie应用所在的源服务器URL子集的路径，cookie的最长有效时间。

SetCookie接口代表由源服务器发送给HTTP代理的响应中的头部信息Set-Cookie来维持一个对话状态。SetCookie2接口和指定的Set-Cookie2方法扩展了SetCookie。

SetCookie接口和额外的如获取原始cookie属性的能力，就像它们由源服务器指定的客户端特定功能扩展了Cookie接口。这对生成Cookie头部很重要，因为一些cookie规范需要。Cookie头部应该包含在Set-Cookie或Set-Cookie2头部中指定的特定属性。

3.1.1 Cookie版本

Cookie兼容网景公司的草案标准，但是版本0被认为是不符合官方规范的。符合标准的cookie的期望版本是1。HttpClient可以处理基于不同版本的cookie。

这里有一个重新创建网景公司草案cookie示例：

```
BasicClientCookie netscapeCookie = new BasicClientCookie("name",
 "value");
netscapeCookie.setVersion(0);
netscapeCookie.setDomain(".mycompany.com");
netscapeCookie.setPath("/");
```

这是一个重新创建标准cookie的示例。要注意符合标准的cookie必须保留由源服务器发送的所有属性：

```
BasicClientCookie stdCookie = new BasicClientCookie("name", "val
ue");
stdCookie.setVersion(1);
stdCookie.setDomain(".mycompany.com");
stdCookie.setPath("/");
stdCookie.setSecure(true);
// 精确设置由服务器发送的属性
stdCookie.setAttribute(ClientCookie.VERSION_ATTR, "1");
stdCookie.setAttribute(ClientCookie.DOMAIN_ATTR, ".mycompany.com
");
```

这是一个重新创建Set-Cookie2兼容cookie的实例。要注意符合标准的cookie必须保留由源服务器发送的所有属性：

```
BasicClientCookie2 stdCookie = new BasicClientCookie2("name", "v
alue");
stdCookie.setVersion(1);
stdCookie.setDomain(".mycompany.com");
stdCookie.setPorts(new int[] {80,8080});
stdCookie.setPath("/");
stdCookie.setSecure(true);
// 精确设置由服务器发送的属性
stdCookie.setAttribute(ClientCookie.VERSION_ATTR, "1");
stdCookie.setAttribute(ClientCookie.DOMAIN_ATTR, ".mycompany.com
");
stdCookie.setAttribute(ClientCookie.PORT_ATTR, "80,8080");
```

3.2 Cookie规范

CookieSpec接口代表了cookie管理的规范。Cookie管理规范希望如下几点：

- 解析的Set-Cookie规则还有可选的Set-Cookie2头部信息。
- 验证解析cookie的规则。
- 格式化给定主机的Cookie头部信息，原始端口和路径。

HttpClient附带了一些CookieSpec的实现：

- 网景公司草案：这个规范符合由网景通讯发布的原始草案规范。应当避免，除非有绝对的必要去兼容遗留代码。
- RFC 2109：官方HTTP状态管理规范并取代的老版本，被RFC 2965取代。
- RFC 2965：官方HTTP状态管理规范。
- 浏览器兼容性：这个实现努力去密切模仿（mis）通用Web浏览器应用程序的实现。比如微软的Internet Explorer和Mozilla的FireFox浏览器。
- 最佳匹配：'Meta'（元）cookie规范采用了一些基于又HTTP响应发送的cookie

格式的cookie策略。它基本上聚合了以上所有的实现到以一个类中。

强烈建议使用Best Match策略，让HttpClient在运行时基于执行上下文采用一些合适的兼容等级。

3.3 HTTP cookie和状态管理参数

这些是用于定制HTTP状态管理和独立的cookie规范行为的参数。

- 'http.protocol.cookie-datepatterns'：定义了用于解析非标准的expires属性的合法日期格式。只是对兼容不符合规定的，仍然使用网景公司草案定义的expires而不使用标准的max-age属性服务器需要。这个参数期望得到一个java.util.Collection类型的值。集合元素必须是java.lang.String类型，来兼容java.text.SimpleDateFormat的语法。如果这个参数没有被设置，那么默认的选择就是CookieSpec实现规范的值。要注意这个参数的应用。
- 'http.protocol.single-cookie-header'：定义了是否cookie应该强制到一个独立的Cookie请求头部信息中。否则，每个cookie就被当作分离的Cookie头部信息来格式化。这个参数期望得到一个java.lang.Boolean类型的值。如果这个参数没有被设置，那么默认的选择就是CookieSpec实现规范的值。要注意这个参数仅仅严格应用于cookie规范（RFC 2109和RFC 2965）。浏览器兼容性和网景公司草案策略将会放置所有的cookie到一个请求头部信息中。
- 'http.protocol.cookie-policy'：定义了用于HTTP状态管理的cookie规范的名字。这个参数期望得到一个java.lang.String类型的值。如果这个参数没有被设置，那么合法的日期格式就是CookieSpec实现规范的值。

3.4 Cookie规范注册表

HttpClient使用CookieSpecRegistry类维护一个可用的cookie规范注册表。下面的规范对于每个默认都是注册过的：

- 兼容性：浏览器兼容性（宽松策略）。
- 网景：网景公司草案。
- rfc2109：RFC 2109（过时的严格策略）。
- rfc2965：RFC 2965（严格策略的标准符合）。
- best-match：最佳匹配meta（元）策略。

3.5 选择cookie策略

Cookie策略可以在HTTP客户端被设置，如果需要，在HTTP请求级重写。

```
HttpClient httpclient = new DefaultHttpClient();
// 对每个默认的强制严格cookie策略
httpclient.getParams().setParameter(
    ClientPNames.COOKIE_POLICY, CookiePolicy.RFC_2965);
HttpGet httpget = new HttpGet("http://www.broken-server.com/");
// 对这个请求覆盖默认策略
httpget.getParams().setParameter(
    ClientPNames.COOKIE_POLICY, CookiePolicy.BROWSER_COMPATIBILI
TY);
```

3.6 定制cookie策略

为了实现定制cookie策略，我们应该创建CookieSpec接口的定制实现类，创建一个CookieSpecFactory实现来创建和初始化定制实现的实例并和HttpClient注册这个工厂。一旦定制实现被注册了，它可以和标准的cookie实现有相同的活性。

```
CookieSpecFactory csf = new CookieSpecFactory() {
    public CookieSpec newInstance(HttpParams params) {
        return new BrowserCompatSpec() {
            @Override
            public void validate(Cookie cookie, CookieOrigin ori
gin)
            throws MalformedCookieException {
                // 这相当简单
            }
        };
    }
};
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.getCookieSpecs().register("easy", csf);
httpclient.getParams().setParameter(
    ClientPNames.COOKIE_POLICY, "easy");
```

3.7 Cookie持久化

HttpClient可以和任意物理表示的实现了CookieStore接口的持久化cookie存储一起使用。默认的CookieStore实现称为BasicClientCookie，这是凭借java.util.ArrayList的一个简单实现。在BasicClientCookie对象中存储的cookie当容器对象被垃圾回收机制回收时会丢失。如果需要，用户可以提供更复杂的实现。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
// 创建一个本地的cookie store实例
CookieStore cookieStore = new MyCookieStore();
// 如果需要填充cookie
BasicClientCookie cookie = new BasicClientCookie("name", "value"
);
cookie.setVersion(0);
cookie.setDomain(".mycompany.com");
cookie.setPath("/");
cookieStore.addCookie(cookie);
// 设置存储
httpclient.setCookieStore(cookieStore);
```

3.8 HTTP状态管理和执行上下文

在HTTP请求执行的过程中，HttpClient添加了下列和状态管理相关的对象到执行上下文中：

- `http.cookiespec-registry` ：CookieSpecRegistry实例代表了实际的cookie规范注册表。这个属性的值设置在本地内容中，优先于默认的。
- `http.cookie-spec` ：CookieSpec实例代表眞实的cookie规范。
- `http.cookie-origin` ：CookieOrigin实例代表了眞实的源服务器的详细信息。
- `http.cookie-store` ：CookieStore实例代表了眞实的cookie存储。设置在本地内容中的这个属性的值优先于默认的。

本地的HttpContext对象可以被用来定制HTTP状态管理内容，先于请求执行或在请求执行之后检查它的状态：

```
HttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget, localContext
);
CookieOrigin cookieOrigin = (CookieOrigin) localContext.getAttri
bute(
    ClientContext.COOKIE_ORIGIN);
System.out.println("Cookie origin: " + cookieOrigin);
CookieSpec cookieSpec = (CookieSpec) localContext.getAttribute(
    ClientContext.COOKIE_SPEC);
System.out.println("Cookie spec used: " + cookieSpec);
```

3.9 每个用户/线程的状态管理

我们可以使用独立的本地执行上下文来实现对每个用户（或每个线程）状态的管理。定义在本地内容中的cookie规范注册表和cookie存储将会优先于设置在HTTP客户端级别中默认的那些。

```
HttpClient httpclient = new DefaultHttpClient();
// 创建cookie store的本地实例
CookieStore cookieStore = new BasicCookieStore();
// 创建本地的HTTP内容
HttpContext localContext = new BasicHttpContext();
// 绑定定制的cookie store到本地内容中
localContext.setAttribute(ClientContext.COOKIE_STORE, cookieStor
e);
HttpGet httpget = new HttpGet("http://www.google.com/");
// 作为参数传递本地内容
HttpResponse response = httpclient.execute(httpget, localContext
)
```

第四章 HTTP认证

HttpClient提供对由HTTP标准规范定义的认证模式的完全支持。HttpClient的认证框架可以扩展支持非标准的认证模式，比如NTLM和SPNEGO。

4.1 用户凭证

任何用户身份验证的过程都需要一组可以用于建立用户身份的凭据。用户凭证的最简单的形式可以仅仅是用户名/密码对。UsernamePasswordCredentials代表了一组包含安全规则和明文密码的凭据。这个实现对由HTTP标准规范中定义的标准认证模式是足够的

```
UsernamePasswordCredentials creds = new UsernamePasswordCredenti
als("user", "pwd");
System.out.println(creds.getUserPrincipal().getName());
System.out.println(creds.getPassword());
```

输出内容为：

```
user
pwd
```

NTCredentials是微软Windows指定的实现，它包含了除了用户名/密码对外，一组额外的Windows指定的属性，比如用户域名的名字，比如在微软的Windows网络中，相同的用户使用不同设置的认证可以属于不同的域。

```
NTCredentials creds = new NTCredentials("user", "pwd", "workstat
ion", "domain");
System.out.println(creds.getUserPrincipal().getName());
System.out.println(creds.getPassword());
```

输出内容为：

```
DOMAIN/user
pwd
```

4.2 认证模式

AuthScheme接口代表了抽象的，面向挑战-响应的认证模式。一个认证模式期望支持如下的功能：

- 解析和处理由目标服务器在对受保护资源请求的响应中发回的挑战。
- 提供处理挑战的属性：认证模式类型和它的参数，如果可用，比如这个认证模型可应用的领域。
- 对给定的凭证组和HTTP请求对响应真实认证挑战生成认证字符串。

要注意认证模式可能是有状态的，涉及一系列的挑战-响应交流。HttpClient附带了一些AuthScheme实现：

- Basic（基本）：Basic认证模式定义在RFC 2617中。这个认证模式是不安全的，因为凭据以明文形式传送。尽管它不安全，如果用在和TLS/SSL加密的组合中，Basic认证模式是完全够用的。
- Digest（摘要）：Digest认证模式定义在RFC 2617中。Digest认证模式比Basic有显著的安全提升，对不想通过TLS/SL加密在完全运输安全上开销的应用程序来说也是很好的选择。
- NTLM：NTLM是一个由微软开发的优化Windows平台的专有认证模式。NTLM被认为是比Digest更安全的模式。这个模式需要外部的NTLM引擎来工作。要获取更多详情请参考包含在HttpClient发布包中的NTLM_SUPPORT.txt文档。

4.3 HTTP认证参数

有一些可以用于定制HTTP认证过程和独立认证模式行为的参数：

- `http.protocol.handle-authentication`：定义了是否认证应该被自动处理。这个参数期望的得到一个java.lang.Boolean类型的值。如果这个参数没有被设置，HttpClient将会自动处理认证。
- `http.auth.credential-charset`：定义了当编码用户凭证时使用的字符集。这个参数期望得到一个java.lang.String类型的值。如果这个参数没有被设置，那么就会使用US-ASCII。

4.4 认证模式注册表

HttpClient使用AuthSchemeRegistry类维护一个可用的认证模式的注册表。对于每个默认的下面的模式是注册过的：

- Basic：基本认证模式
- Digest：摘要认证模式

请注意NTLM模式没有对每个默认的进行注册。NTLM不能对每个默认开启是应为许可和法律上的原因。要获取更详细的关于如何开启NTLM支持的内容请看这部分。

4.5 凭据提供器

凭据提供器意来维护一组用户凭据，还有能够对特定认证范围生产用户凭据。认证范围包括主机名，端口号，领域名称和认证模式名称。当使用凭据提供器来注册凭据时，我们可以提供一个通配符（任意主机，任意端口，任意领域，任意模式）来

替代确定的属性值。如果直接匹配没有发现，凭据提供器期望被用来发现最匹配的特定范围。

HttpClient可以和任意实现了CredentialsProvider接口的凭据提供器的物理代表一同工作。默认的CredentialsProvider实现被称为BasicCredentialsProvider，它是简单的凭借java.util.HashMap的实现。

```
CredentialsProvider credsProvider = new BasicCredentialsProvider
();
credsProvider.setCredentials(
    new AuthScope("somehost", AuthScope.ANY_PORT),
    new UsernamePasswordCredentials("u1", "p1"));
credsProvider.setCredentials(
    new AuthScope("somehost", 8080),
    new UsernamePasswordCredentials("u2", "p2"));
credsProvider.setCredentials(
    new AuthScope("otherhost", 8080, AuthScope.ANY_REALM, "ntlm"
),
    new UsernamePasswordCredentials("u3", "p3"));
System.out.println(credsProvider.getCredentials(
    new AuthScope("somehost", 80, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
    new AuthScope("somehost", 8080, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
    new AuthScope("otherhost", 8080, "realm", "basic")));
System.out.println(credsProvider.getCredentials(
    new AuthScope("otherhost", 8080, null, "ntlm")));
```

输出内容为：

```
[principal: u1]
[principal: u2]
null
[principal: u3]
```

4.6 HTTP认证和执行上下文

HttpClient依赖于AuthState类来跟踪关于认证过程状态的详细信息。在HTTP请求执行过程中，HttpClient创建2个AuthState的实例：一个对于目标主机认证，另外一个对于代理认证。如果目标服务器或代理需要用户认证，那么各自的AuthState实例将会被在认证处理过程中使用的AuthScope，AuthScheme和Crednetials来填充。AuthState可以被检查来找出请求的认证是什么类型的，是否匹配AuthScheme的实现，是否凭据提供器对给定的认证范围去找用户凭据。

在HTTP请求执行的过程中，HttpClient添加了下列和认证相关的对象到执行上下文中：

- `http.authscheme-registry`：AuthSchemeRegistry实例代表真实的认证模式注册表。在本地内容中设置的这个属性的值优先于默认的。
- `http.auth.credentials-provider`：CookieSpec实例代表了真实的凭据提供器。在本地内容中设置的这个属性的值优先于默认的。
- `http.auth.target-scope`：AuthState实例代表了真实的目标认证状态。在本地内容中设置的这个属性的值优先于默认的。
- `http.auth.proxy-scope`：AuthState实例代表了真实的代理认证状态。在本地内容中设置的这个属性的值优先于默认的。

本地的HttpContext对象可以用于定制HTTP认证内容，并先于请求执行或在请求被执行之后检查它的状态：

```
HttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget, localContext
);
AuthState proxyAuthState = (AuthState) localContext.getAttribute
(
    ClientContext.PROXY_AUTH_STATE);
System.out.println("Proxy auth scope: " + proxyAuthState.getAuth
Scope());
System.out.println("Proxy auth scheme: " + proxyAuthState.getAut
hScheme());
System.out.println("Proxy auth credentials: " + proxyAuthState.g
etCredentials());
AuthState targetAuthState = (AuthState) localContext.getAttribut
e(
    ClientContext.TARGET_AUTH_STATE);
System.out.println("Target auth scope: " + targetAuthState.getAu
thScope());
System.out.println("Target auth scheme: " + targetAuthState.getA
uthScheme());
System.out.println("Target auth credentials: " + targetAuthState
.getCredentials());
```

4.7 抢占认证

HttpClient不支持开箱的抢占认证，因为滥用或重用不正确的抢占认证可能会导致严重的安全问题，比如将用户凭据以明文形式发送给未认证的第三方。因此，用户期望评估抢占认证和在它们只能应用程序环境内容安全风险潜在的好处，而且要求使用如协议拦截器的标准HttpClient扩展机制添加对抢占认证的支持。

这是一个简单的协议拦截器，如果没有企图认证，来抢先引入BasicScheme的实例到执行上下文中。请注意拦截器必须在标准认证拦截器之前加入到协议处理链中。

```
HttpRequestInterceptor preemptiveAuth = new HttpRequestIntercept
or() {
    public void process(final HttpRequest request,
    final HttpContext context) throws HttpException, IOException
 {
        AuthState authState = (AuthState) context.getAttribute(
            ClientContext.TARGET_AUTH_STATE);
        CredentialsProvider credsProvider = (CredentialsProvider
) context.getAttribute(ClientContext.CREDS_PROVIDER);
        HttpHost targetHost = (HttpHost) context.getAttribute(
            ExecutionContext.HTTP_TARGET_HOST);
        // 如果没有初始化auth模式
        if (authState.getAuthScheme() == null) {
            AuthScope authScope = new AuthScope(
                targetHost.getHostName(),
                targetHost.getPort());
            // 获得匹配目标主机的凭据
            Credentials creds = credsProvider.getCredentials(aut
hScope);
            // 如果发现了，抢先生成BasicScheme
            if (creds != null) {
                authState.setAuthScheme(new BasicScheme());
                authState.setCredentials(creds);
            }
        }
    }
};
DefaultHttpClient httpclient = new DefaultHttpClient();
// 作为第一个拦截器加入到协议链中
httpclient.addRequestInterceptor(preemptiveAuth, 0);
```

4.8 NTLM 认证

当前HttpClient没有提对开箱的NTLM认证模式的支持也可能永远也不会。这个原因是法律上的而不是技术上的。然而，NTLM认证可以使用外部的NTLM引擎比如JCIFS来开启，类库由Samba项目开发，作为它们Windows的交互操作程序套装的一部分。要获取详细内容请参考HttpClient发行包中包含的NTLM_SUPPORT.txt文档。

4.8.1 NTLM连接持久化

NTLM认证模式是在计算开销方面昂贵的多的，而且对标准的Basic和Digest模式的性能影响也很大。这很可能是为什么微软选择NTLM认证模式为有状态的主要原因之一。也就是说，一旦认证通过，用户标识是和连接的整个生命周期相关联的。NTLM连接的状态特性使得连接持久化非常复杂，对于明显的原因，持久化NTLM连接不能被使用不同用户标识的用户重用。标准的连接管理器附带HttpClient是完全能够管理状态连接的。而逻辑相关的，使用同一session和执行上下文为了让它们了

解到当前的用户标识的请求也是极为重要的。否则，HttpClient将会终止对每个基于NTLM保护资源的HTTP请求创建新的HTTP连接。要获取关于有状态的HTTP连接的详细讨论，请参考这个部分。

因为NTLM连接是有状态的，通常建议使用相对简单的方法触发NTLM认证，比如GET或HEAD，而重用相同的连接来执行代价更大的方法，特别是它们包含请求实体，比如POST或PUT。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
NTCredentials creds = new NTCredentials("user", "pwd", "myworkst
ation", "microsoft.com");
httpclient.getCredentialsProvider().setCredentials(AuthScope.ANY
, creds);
HttpHost target = new HttpHost("www.microsoft.com", 80, "http");
// 保证相同的内容来用于执行逻辑相关的请求
HttpContext localContext = new BasicHttpContext();
// 首先执行简便的方法。这会触发NTLM认证
HttpGet httpget = new HttpGet("/ntlm-protected/info");
HttpResponse response1 = httpclient.execute(target, httpget, loc
alContext);
HttpEntity entity1 = response1.getEntity();
if (entity1 != null) {
    entity1.consumeContent();
}
//之后使用相同的内容（和连接）执行开销大的方法。
HttpPost httppost = new HttpPost("/ntlm-protected/form");
httppost.setEntity(new StringEntity("lots and lots of data"));
HttpResponse response2 = httpclient.execute(target, httppost, lo
calContext);
HttpEntity entity2 = response2.getEntity();
if (entity2 != null) {
    entity2.consumeContent();
}
```

第五章 HTTP客户端服务

5.1 HttpClient门面

HttpClient接口代表了最重要的HTTP请求执行的契约。它没有在请求执行处理上强加限制或特殊细节，而在连接管理，状态管理，认证和处理重定向到具体实现上留下了细节。这应该使得很容易使用额外的功能，比如响应内容缓存来装饰接口。

DefaultHttpClient是HttpClient接口的默认实现。这个类扮演了很多特殊用户程序或策略接口实现负责处理特定HTTP协议方面，比如重定向到处理认证或做出关于连接持久化和保持活动的持续时间决定的门面。这使得用户可以选择使用定制，具体程序等来替换某些方面默认实现。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setKeepAliveStrategy(new DefaultConnectionKeepAliveSt
rategy() {
    @Override
    public long getKeepAliveDuration(HttpResponse response,
    HttpContext context) {
        long keepAlive = super.getKeepAliveDuration(response, co
ntext);
        if (keepAlive == -1) {
            // 如果keep-alive值没有由服务器明确设置，那么保持连接持续5秒
。
            keepAlive = 5000;
        }
        return keepAlive;
    }
});
```

DefaultHttpClient也维护一组协议拦截器，意在处理即将离开的请求和即将到达的响应，而且提供管理那些拦截器的方法。新的协议拦截器可以被引入到协议处理器链中，或在需要时从中移除。内部的协议拦截器存储在一个简单的java.util.ArrayList中。它们以被加入到list中的自然顺序来执行。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.removeRequestInterceptorByClass(RequestUserAgent.clas
s);
httpclient.addRequestInterceptor(new HttpRequestInterceptor() {
    public void process(
        HttpRequest request, HttpContext context)
    throws HttpException, IOException {
        request.setHeader(HTTP.USER_AGENT, "My-own-client");
    }
});
```

DefaultHttpClient是线程安全的。建议相同的这个类的实例被重用于多个请求的执行。当一个DefaultHttpClient实例不再需要而且要脱离范围时，和它关联的连接管理器必须调用ClientConnectionManager#shutdown()方法关闭。

```
HttpClient httpclient = new DefaultHttpClient();
// 做些有用的事
httpclient.getConnectionManager().shutdown();
```

5.2 HttpClient参数

这些是可以用于定制默认HttpClient实现行为的参数：

- `http.protocol.handle-redirects`：定义了重定向是否应该自动处理。这个参数期望得到一个java.lang.Boolean类型的值。如果这个参数没有被设置，HttpClient将会自动处理重定向。
- `http.protocol.reject-relative-redirect`：定义了是否相对的重定向应该被拒绝。HTTP规范需要位置值是一个绝对URI。这个参数期望得到一个java.lang.Boolean类型的值。如果这个参数没有被设置，那么就允许相对重定向。
- `http.protocol.max-redirects`：定义了要遵循重定向的最大数量。这个重定向数字的限制意在防止由破碎的服务器端脚本引发的死循环。这个参数期望得到一个java.lang.Integer类型的值。如果这个参数没有被设置，那么只允许不多余100次重定向。
- `http.protocol.allow-circular-redirects`：定义环形重定向（重定向到相同路径）是否被允许。HTTP规范在环形重定向没有足够清晰的允许表述，因此这作为可选的是可以开启的。这个参数期望得到一个java.lang.Boolean类型的值。如果这个参数没有被设置，那么环形重定向就不允许。
- `http.connection-manager.factory-class-name`：定义了默认的ClientConnectionManager实现的类型。这个参数期望得到一个java.lang.String类型的值。如果这个参数没有被设置，对于每个默认的将使用SingleClientConnManager。
- `http.virtual-host`：定义了在头部信息Host中使用的虚拟主机名称，来代替物理主机名称。这个参数期望得到一个HttpHost类型的值。如果这个参数没有被设置，那么将会使用目标主机的名称或IP地址。
- `http.default-headers`：定义了每次请求默认发送的头部信息。这个参数期望得到一个包含Header对象的java.util.Collection类型值。
- `http.default-host`：定义了默认主机。如果目标主机没有在请求URI（相对URI）中明确指定，那么就使用默认值。这个参数期望得到一个HttpHost类型的值。

5.3 自动重定向处理

HttpClient处理所有类型的自动重定向，除了那些由HTTP规范明令禁止的，比如需要用户干预的。参考其它（状态码303）POST和PUT请求重定向转换为由HTTP规范需要的GET请求。

5.4 HTTP客户端和执行上下文

DefaultHttpClient将HTTP请求视为不变的对象，也从来不会假定在请求执行期间改变。相反，它创建了一个原请求对象私有的可变副本，副本的属性可以基于执行上下文来更新。因此，如目标主键和请求URI的final类型的请求参数可以在请求执行之后，由检查本地HTTP上下文来决定。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget, localContext
);
HttpHost target = (HttpHost) localContext.getAttribute(
    ExecutionContext.HTTP_TARGET_HOST);
HttpUriRequest req = (HttpUriRequest) localContext.getAttribute(
    ExecutionContext.HTTP_REQUEST);
System.out.println("Target host: " + target);
System.out.println("Final request URI: " + req.getURI());
System.out.println("Final request method: " + req.getMethod());
```

第六章 高级主题

6.1 自定义客户端连接

在特定条件下，也许需要来定制HTTP报文通过线路传递，越过了可能使用的HTTP参数来处理非标准不兼容行为的方式。比如，对于Web爬虫，它可能需要强制HttpClient接受格式错误的响应头部信息，来抢救报文的内容。

通常插入一个自定义的报文解析器的过程或定制连接实现需要几个步骤：

提供一个自定义LineParser/LineFormatter接口实现。如果需要，实现报文解析/格式化逻辑。

```
class MyLineParser extends BasicLineParser {
    @Override
    public Header parseHeader(
    final CharArrayBuffer buffer) throws ParseException {
        try {
            return super.parseHeader(buffer);
        } catch (ParseException ex) {
            // 压制ParseException异常
            return new BasicHeader("invalid", buffer.toString())
;
        }
    }
}
```

提过一个自定义的OperatedClientConnection实现。替换需要自定义的默认请求/响应解析器，请求/响应格式化器。如果需要，实现不同的报文写入/读取代码。

```
class MyClientConnection extends DefaultClientConnection {
    @Override
    protected HttpMessageParser createResponseParser(
        final SessionInputBuffer buffer,
        final HttpResponseFactory responseFactory,
        final HttpParams params) {
        return new DefaultResponseParser(buffer,
        new MyLineParser(),responseFactory,params);
    }
}
```

为了创建新类的连接，提供一个自定义的ClientConnectionOperator接口实现。如果需要，实现不同的套接字初始化代码。

```
class MyClientConnectionOperator extends
DefaultClientConnectionOperator {
    public MyClientConnectionOperator(
        final SchemeRegistry sr) {
        super(sr);
    }
    @Override
    public OperatedClientConnection createConnection() {
        return new MyClientConnection();
    }
}
```

为了创建新类的连接操作，提供自定义的ClientConnectionManager接口实现。

```
class MyClientConnManager extends SingleClientConnManager {
    public MyClientConnManager(
        final HttpParams params,
        final SchemeRegistry sr) {
        super(params, sr);
    }
    @Override
    protected ClientConnectionOperator createConnectionOperator(
    final SchemeRegistry sr) {
        return new MyClientConnectionOperator(sr);
    }
}
```

6.2 有状态的HTTP连接

HTTP规范假设session状态信息通常是以HTTP cookie格式嵌入在HTTP报文中的，因此HTTP连接通常是无状态的，这个假设在现实生活中通常是不对的。也有一些情况，当HTTP连接使用特定的用户标识或特定的安全上下文来创建时，因此不能和其它用户共享，只能由该用户重用。这样的有状态的HTTP连接的示例就是NTLM认证连接和使用客户端证书认证的SSL连接。

6.2.1 用户令牌处理器

HttpClient依赖UserTokenHandler接口来决定给定的执行上下文是否是用户指定的。如果这个上下文是用户指定的或者如果上下文没有包含任何资源或关于当前用户指定详情而是null，令牌对象由这个处理器返回，期望唯一地标识当前的用户。用户令牌将被用来保证用户指定资源不会和其它用户来共享或重用。

如果它可以从给定的执行上下文中来获得，UserTokenHandler接口的默认实现是使用主类的一个实例来代表HTTP连接的状态对象。UserTokenHandler将会使用基于如NTLM或开启的客户端认证SSL会话认证模式的用户的主连接。如果二者都不可用，那么就不会返回令牌。

如果默认的不能满足它们的需要，用户可以提供一个自定义的实现：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
httpclient.setUserTokenHandler(new UserTokenHandler() {
    public Object getUserToken(HttpContext context) {
        return context.getAttribute("my-token");
}
});
```

6.2.2 用户令牌和执行上下文

在HTTP请求执行的过程中，HttpClient添加了下列和用户标识相关的对象到执行上下文中：

`http.user-token`：对象实例代表真实的用户标识，通常期望Principle接口的实例。

我们可以在请求被执行后，通过检查本地HTTP上下文的内容，发现是否用于执行请求的连接是有状态的。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext = new BasicHttpContext();
HttpGet httpget = new HttpGet("http://localhost:8080/");
HttpResponse response = httpclient.execute(httpget, localContext
);
HttpEntity entity = response.getEntity();
if (entity != null) {
    entity.consumeContent();
}
Object userToken = localContext.getAttribute(ClientContext.USER_
TOKEN);
System.out.println(userToken);
```

6.2.2.1 持久化有状态的连接

请注意带有状态对象的持久化连接仅当请求被执行时，相同状态对象被绑定到执行上下文时可以被重用。所以，保证相同上下文重用于执行随后的相同用户，或用户令牌绑定到之前请求执行上下文的HTTP请求是很重要的。

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpContext localContext1 = new BasicHttpContext();
HttpGet httpget1 = new HttpGet("http://localhost:8080/");
HttpResponse response1 = httpclient.execute(httpget1, localConte
xt1);
HttpEntity entity1 = response1.getEntity();
if (entity1 != null) {
    entity1.consumeContent();
}
Principal principal = (Principal) localContext1.getAttribute(
    ClientContext.USER_TOKEN);
HttpContext localContext2 = new BasicHttpContext();
localContext2.setAttribute(ClientContext.USER_TOKEN, principal);
HttpGet httpget2 = new HttpGet("http://localhost:8080/");
HttpResponse response2 = httpclient.execute(httpget2, localConte
xt2);
HttpEntity entity2 = response2.getEntity();
if (entity2 != null) {
    entity2.consumeContent();
}
```

MyBatis 中文文档 3.4

来源：MyBatis中文文档

参考文档

简介

什么是 MyBatis ？

MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以对配置和原生Map使用简单的 XML 或注解，将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java对象)映射成数据库中的记录。

帮助改进文档...

不管你以何种方式发现了文档的不足，或是丢失对某一特性的描述，那么你能做的最好的事情莫过于去研究它并把文档写出来。

该文档 xdoc 格式的源码文件可通过项目的 Git 代码库来获取。Fork 该源码库，做出更新，然后提交一个 pull request 吧。

你将成为本文档的最佳作者，MyBatis 的用户定会过来查阅的。

当前的国际化版本

MyBatis 的其他语言版本：

- English
- Espa?ol
- 日本語
- ???
- 简体中文

你想使用本地语言来了解MyBatis吗？那就将它翻译成你的母语并提供给我们吧！

入门

安装

要使用 MyBatis， 只需将 `mybatis-x.x.x.jar` 文件置于 classpath 中即可。

如果使用 Maven 来构建项目，则需将下面的 dependency 代码置于 pom.xml 文件中：

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>
```

从 XML 中构建 SqlSessionFactory

每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为中心的。SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先定制的 Configuration 的实例构建出 SqlSessionFactory 的实例。

从 XML 文件中构建 SqlSessionFactory 的实例非常简单，建议使用类路径下的资源文件进行配置。但是也可以使用任意的输入流(InputStream)实例，包括字符串形式的文件路径或者 file:// 的 URL 形式的文件路径来配置。MyBatis 包含一个名叫 Resources 的工具类，它包含一些实用方法，可使从 classpath 或其他位置加载资源文件更加容易。

```
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource
);
sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputSt
ream);
```

XML 配置文件（configuration XML）中包含了对 MyBatis 系统的核心设置，包含获取数据库连接实例的数据源（DataSource）和决定事务范围和控制方式的事务管理器（TransactionManager）。XML 配置文件的详细内容后面再探讨，这里先给出一个简单的示例：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username" value="${username}"/>
        <property name="password" value="${password}"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="org/mybatis/example/BlogMapper.xml"/>
  </mappers>
</configuration>
```

当然，还有很多可以在XML 文件中进行配置，上面的示例指出的则是最关键的部分。要注意 XML 头部的声明，用来验证 XML 文档正确性。environment 元素体中包含了事务管理和连接池的配置。mappers 元素则是包含一组 mapper 映射器（这些 mapper 的 XML 文件包含了 SQL 代码和映射定义信息）。

不使用 XML 构建 SqlSessionFactory

如果你更愿意直接从 Java 程序而不是 XML 文件中创建 configuration，或者创建你自己的 configuration 构建器，MyBatis 也提供了完整的配置类，提供所有和 XML 文件相同功能的配置项。

```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSource(
);
TransactionFactory transactionFactory = new JdbcTransactionFacto
ry();
Environment environment = new Environment("development", transac
tionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.addMapper(BlogMapper.class);
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuild
er().build(configuration);
```

注意该例中，configuration 添加了一个映射器类（mapper class）。映射器类是 Java 类，它们包含 SQL 映射语句的注解从而避免了 XML 文件的依赖。不过，由于 Java 注解的一些限制加之某些 MyBatis 映射的复杂性，XML 映射对于大多数高级映射（比如：嵌套 Join 映射）来说仍然是必须的。有鉴于此，如果存在一个对等

的 XML 配置文件的话，MyBatis 会自动查找并加载它（这种情况下，BlogMapper.xml 将会基于类路径和 BlogMapper.class 的类名被加载进来）。具体细节稍后讨论。

从 SqlSessionFactory 中获取 SqlSession

既然有了 SqlSessionFactory ，顾名思义，我们就可以从中获得 SqlSession 的实例了。SqlSession 完全包含了面向数据库执行 SQL 命令所需的所有方法。你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。例如：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  Blog blog = (Blog) session.selectOne("org.mybatis.example.Blog
Mapper.selectBlog", 101);
} finally {
  session.close();
}
```

诚然这种方式能够正常工作，并且对于使用旧版本 MyBatis 的用户来说也比较熟悉，不过现在有了一种更直白的方式。使用对于给定语句能够合理描述参数和返回值的接口（比如说BlogMapper.class），你现在不但可以执行更清晰和类型安全的代码，而且还不用担心易错的字符串字面值以及强制类型转换。

例如：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  Blog blog = mapper.selectBlog(101);
} finally {
  session.close();
}
```

现在我们来探究一下这里到底是怎么执行的。

探究已映射的 SQL 语句

现在，或许你很想知道 SqlSession 和 Mapper 到底执行了什么操作，而 SQL 语句映射是个相当大的话题，可能会占去文档的大部分篇幅。不过为了让你能够了解个大概，这里会给出几个例子。

在上面提到的两个例子中，一个语句应该是通过 XML 定义，而另外一个则是通过注解定义。先看 XML 定义这个，事实上 MyBatis 提供的全部特性可以利用基于 XML 的映射语言来实现，这使得 MyBatis 在过去的数年间得以流行。如果你以前用过 MyBatis，这个概念应该会比较熟悉。不过 XML 映射文件已经有了很多的改进，随着文档的进行会愈发清晰。这里给出一个基于 XML 映射语句的示例，它应该可以满足上述示例中 SqlSession 的调用。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

对于这个简单的例子来说似乎有点小题大做了，但实际上它是非常轻量级的。在一个 XML 映射文件中，你想定义多少个映射语句都是可以的，这样下来，XML 头部和文档类型声明占去的部分就显得微不足道了。文件的剩余部分具有很好的自解释性。在命名空间“com.mybatis.example.BlogMapper”中定义了一个名为“selectBlog”的映射语句，这样它就允许你使用指定的完全限定名“org.mybatis.example.BlogMapper.selectBlog”来调用映射语句，就像上面的例子中做的那样：

```
Blog blog = (Blog) session.selectOne("org.mybatis.example.BlogMa
pper.selectBlog", 101);
```

你可能注意到这和使用完全限定名调用 Java 对象的方法是相似的，之所以这样做是有原因的。这个命名可以直接映射到在命名空间中同名的 Mapper 类，并在已映射的 select 语句中的名字、参数和返回类型匹配成方法。这样你就可以向上面那样很容易地调用这个对应 Mapper 接口的方法。不过让我们再看一遍下面的例子：

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

第二种方法有很多优势，首先它不是基于字符串常量的，就会更安全；其次，如果你的 IDE 有代码补全功能，那么你可以在有了已映射 SQL 语句的基础之上利用这个功能。

提示命名空间的一点注释

命名空间（**Namespaces**）在之前版本的 MyBatis 中是可选的，容易引起混淆因此是没有益处的。现在的命名空间则是必须的，目的是希望能比只是简单的使用更长的完全限定名来区分语句更进一步。

命名空间使得你所见到的接口绑定成为可能，尽管你觉得这些东西未必用得上，你还是应该遵循这里的规定以防哪天你改变了主意。出于长远考虑，使用命名空间，并将它置于合适的 Java 包命名空间之下，你将拥有一份更加整洁的代码并提高了 MyBatis 的可用性。

命名解析：为了减少输入量，MyBatis 对所有的命名配置元素（包括语句，结果映射，缓存等）使用了如下的命名解析规则。

- 完全限定名（比如“com.mypackage.MyMapper.selectAllThings”）将被直接查找并且找到即用。
- 短名称（比如“selectAllThings”）如果全局唯一也可以作为一个单独的引用。如果不唯一，有两个或两个以上的相同名称（比如“com.foo.selectAllThings”和“com.bar.selectAllThings”），那么使用时就会收到错误报告说短名称是不唯一的，这种情况下就必须使用完全限定名。

对于像 BlogMapper 这样的映射器类（Mapper class）来说，还有另一招来处理。它们的映射的语句可以不需要用 XML 来做，取而代之的是可以使用 Java 注解。比如，上面的 XML 示例可被替换如下：

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

对于简单语句来说，注解使代码显得更加简洁，然而 Java 注解对于稍微复杂的语句就会力不从心并且会显得更加混乱。因此，如果你需要做很复杂的事情，那么最好使用 XML 来映射语句。

选择何种方式以及映射语句的定义的一致性对你来说有多重要这些完全取决于你和你的团队。换句话说，永远不要拘泥于一种方式，你可以很轻松的在基于注解和 XML 的语句映射方式间自由移植和切换。

范围（Scope）和生命周期

理解我们目前已经讨论过的不同范围和生命周期类是至关重要的，因为错误的使用会导致非常严重的并发问题。

提示 对象生命周期和依赖注入框架

依赖注入框架可以创建线程安全的、基于事务的 SqlSession 和映射器（mapper）并将它们直接注入到你的 bean 中，因此可以直接忽略它们的生命周期。如果对如何通过依赖注入框架来使用 MyBatis 感兴趣可以研究一下 MyBatis-Spring 或 MyBatis-Guice 两个子项目。

SqlSessionFactoryBuilder

这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。因此 SqlSessionFactoryBuilder 实例的最佳范围是方法范围（也就是局部方法变量）。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但是最好还是不要让其一直存在以保证所有的 XML 解析资源开放给更重要的事情。

SqlSessionFactory

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由对它进行清除或重建。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏味道（bad smell）”。因此 SqlSessionFactory 的最佳范围是应用范围。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

SqlSession

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的范围是请求或方法范围。绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。也绝不能将 SqlSession 实例的引用放在任何类型的管理范围中，比如 Serlvet 架构中的 HttpSession。如果你现在正在使用一种 Web 框架，要考虑 SqlSession 放在一个和 HTTP 请求对象相似的范围中。换句话说，每次收到的 HTTP 请求，就可以打开一个 SqlSession，返回一个响应，就关闭它。这个关闭操作是很重要的，你应该把这个关闭操作放到 finally 块中以确保每次都能执行关闭。下面的示例就是一个确保 SqlSession 关闭的标准模式：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  // do work
} finally {
  session.close();
}
```

在你的所有的代码中一致性地使用这种模式来保证所有数据库资源都能被正确地关闭。

映射器实例（Mapper Instances）

映射器是创建用来绑定映射语句的接口。映射器接口的实例是从 SqlSession 中获得的。因此从技术层面讲，映射器实例的最大范围是和 SqlSession 相同的，因为它们都是从 SqlSession 里被请求的。尽管如此，映射器实例的最佳范围是方法范围。也就是说，映射器实例应该在调用它们的方法中被请求，用过之后即可废弃。并不需要显式地关闭映射器实例，尽管在整个请求范围（request scope）保持映射器实例也不会有什么问题，但是很快你会发现，像 SqlSession 一样，在这个范围上管理太多的资源的话会难于控制。所以要保持简单，最好把映射器放在方法范围（method scope）内。下面的示例就展示了这个实践：

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  // do work
} finally {
  session.close();
}
```

XML 映射配置文件

MyBatis 的配置文件包含了影响 MyBatis 行为甚深的设置（settings）和属性（properties）信息。文档的顶层结构如下：

- configuration 配置
    - properties 属性
    - settings 设置
    - typeAliases 类型命名
    - typeHandlers 类型处理器
    - objectFactory 对象工厂
    - plugins 插件
    - environments 环境
        - environment 环境变量
            - transactionManager 事务管理器
            - dataSource 数据源
    - databaseIdProvider 数据库厂商标识
    - mappers 映射器

properties

这些属性都是可外部配置且可动态替换的，既可以在典型的 Java 属性文件中配置，亦可通过 properties 元素的子元素来传递。例如：

```
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

其中的属性就可以在整个配置文件中使用来替换需要动态配置的属性值。比如:

```
<dataSource type="POOLED">
  <property name="driver" value="${driver}"/>
  <property name="url" value="${url}"/>
  <property name="username" value="${username}"/>
  <property name="password" value="${password}"/>
</dataSource>
```

这个例子中的 username 和 password 将会由 properties 元素中设置的相应值来替换。 driver 和 url 属性将会由 config.properties 文件中对应的值来替换。这样就为配置提供了诸多灵活选择。

属性也可以被传递到 SqlSessionBuilder.build()方法中。例如：

```
SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reade
r, props);

// ... or ...

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reade
r, environment, props);
```

如果属性在不只一个地方进行了配置，那么 MyBatis 将按照下面的顺序来加载：

- 在 properties 元素体内指定的属性首先被读取。
- 然后根据 properties 元素中的 resource 属性读取类路径下属性文件或根据 url 属性指定的路径读取属性文件，并覆盖已读取的同名属性。
- 最后读取作为方法参数传递的属性，并覆盖已读取的同名属性。

因此，通过方法参数传递的属性具有最高优先级，resource/url 属性中指定的配置文件次之，最低优先级的是 properties 属性中指定的属性。

settings

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。下表描述了设置中各项的意图、默认值等。

设置参数	描述	有效值
cacheEnabled	该配置影响的所有映射器中配置的缓存的全局开关。	true | false
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 特定关联关系中可通过设置 `fetchType` 属性来覆盖该项的开关状态。	true | false
aggressiveLazyLoading	当启用时，对任意延迟属性的调用会使带有延迟加载属性的对象完整加载；反之，每种属性将会按需加载。	true | false
multipleResultSetsEnabled	是否允许单一语句返回多结果集（需要兼容驱动）。	true | false

useColumnLabel	使用列标签代替列名。不同的驱动在这方面会有不同的表现， 具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。	true | false
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要驱动兼容。 如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。	true | false
autoMappingBehavior	指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。 FULL 会自动映射任意复杂的结果集（无论是否嵌套）。	NONE, PARTIAL, FULL
defaultExecutorType	配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。	SIMPLE REUSE BATCH
defaultStatementTimeout	设置超时时间，它决定驱动等待数据库响应的秒数。	Any positive integer
defaultFetchSize	Sets the driver a hint as to control fetching size for return results. This	Any positive integer

	parameter value can be override by a query setting.	
safeRowBoundsEnabled	允许在嵌套语句中使用分页（RowBounds）。	true | false
mapUnderscoreToCamelCase	是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A_COLUMN 到经典 Java 属性名 aColumn 的类似映射。	true | false
localCacheScope	MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。	SESSION | STATEMENT
jdbcTypeForNull	当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。	JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER
lazyLoadTriggerMethods	指定哪个对象的方法触发一次延迟加载。	A method name list separated by commas

defaultScriptingLanguage	指定动态 SQL 生成的默认语言。	A type alias or fully qualified class name.
callSettersOnNulls	指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这对于有 Map.keySet() 依赖或 null 值初始化的时候是有用的。注意基本类型（int、boolean等）是不能设置成 null 的。	true | false
logPrefix	指定 MyBatis 增加到日志名称的前缀。	Any String
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING
proxyFactory	指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。	CGLIB | JAVASSIST
vfsImpl	Specifies VFS implementations	Fully qualified class names of custom VFS implementation separated by commas.

一个配置完整的 settings 元素的示例如下：

```
<settings>
  <setting name="cacheEnabled" value="true"/>
  <setting name="lazyLoadingEnabled" value="true"/>
  <setting name="multipleResultSetsEnabled" value="true"/>
  <setting name="useColumnLabel" value="true"/>
  <setting name="useGeneratedKeys" value="false"/>
  <setting name="autoMappingBehavior" value="PARTIAL"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25"/>
  <setting name="defaultFetchSize" value="100"/>
  <setting name="safeRowBoundsEnabled" value="false"/>
  <setting name="mapUnderscoreToCamelCase" value="false"/>
  <setting name="localCacheScope" value="SESSION"/>
  <setting name="jdbcTypeForNull" value="OTHER"/>
  <setting name="lazyLoadTriggerMethods" value="equals,clone,has
hCode,toString"/>
</settings>
```

typeAliases

类型别名是为 Java 类型设置一个短的名字。它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。例如:

```
<typeAliases>
  <typeAlias alias="Author" type="domain.blog.Author"/>
  <typeAlias alias="Blog" type="domain.blog.Blog"/>
  <typeAlias alias="Comment" type="domain.blog.Comment"/>
  <typeAlias alias="Post" type="domain.blog.Post"/>
  <typeAlias alias="Section" type="domain.blog.Section"/>
  <typeAlias alias="Tag" type="domain.blog.Tag"/>
</typeAliases>
```

当这样配置时， `Blog` 可以用在任何使用 `domain.blog.Blog` 的地方。

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean，比如:

```
<typeAliases>
  <package name="domain.blog"/>
</typeAliases>
```

每一个在包 `domain.blog` 中的 Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。 比如 `domain.blog.Author` 的别名为 `author` ；若有注解，则别名为其注解值。看下面的例子：

```
@Alias("author")
public class Author {
    ...
}
```

已经为许多常见的 Java 类型内建了相应的类型别名。它们都是大小写不敏感的，需要注意的是由基本类型名称重复导致的特殊处理。

别名	映射的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

typeHandlers

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时， 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。

类型处理器	Java 类型	JDBC 类型
`BooleanTypeHandler`	`java.lang.Boolean` , `boolean`	数据库兼容的 `BOOLEAN`
`ByteTypeHandler`	`java.lang.Byte` , `byte`	数据库兼容的 `NUMERIC` 或 `BYTE`
`ShortTypeHandler`	`java.lang.Short` , `short`	数据库兼容的 `NUMERIC` 或 `SHORT INTEGER`
`IntegerTypeHandler`	`java.lang.Integer` , `int`	数据库兼容的 `NUMERIC` 或 `INTEGER`
`LongTypeHandler`	`java.lang.Long` , `long`	数据库兼容的 `NUMERIC` 或 `LONG INTEGER`
`FloatTypeHandler`	`java.lang.Float` , `float`	数据库兼容的 `NUMERIC` 或 `FLOAT`
`DoubleTypeHandler`	`java.lang.Double` , `double`	数据库兼容的 `NUMERIC` 或 `DOUBLE`
`BigDecimalTypeHandler`	`java.math.BigDecimal`	数据库兼容的 `NUMERIC` 或 `DECIMAL`
`StringTypeHandler`	`java.lang.String`	`CHAR` , `VARCHAR`
`ClobTypeHandler`	`java.lang.String`	`CLOB` , `LONGVARCHAR`
`NStringTypeHandler`	`java.lang.String`	`NVARCHAR` , `NCHAR`
`NClobTypeHandler`	`java.lang.String`	`NCLOB`
`ByteArrayTypeHandler`	`byte[]`	数据库兼容的字节流类型
`BlobTypeHandler`	`byte[]`	`BLOB` , `LONGVARBINARY`

`DateTypeHandler`	`java.util.Date`	`TIMESTAMP`
`DateOnlyTypeHandler`	`java.util.Date`	`DATE`
`TimeOnlyTypeHandler`	`java.util.Date`	`TIME`
`SqlTimestampTypeHandler`	`java.sql.Timestamp`	`TIMESTAMP`
`SqlDateTypeHandler`	`java.sql.Date`	`DATE`
`SqlTimeTypeHandler`	`java.sql.Time`	`TIME`
`ObjectTypeHandler`	Any	`OTHER` 或未指定类型
`EnumTypeHandler`	Enumeration Type	VARCHAR-任何兼容的字符串类型，存储枚举的名称（而不是索引）
`EnumOrdinalTypeHandler`	Enumeration Type	任何兼容的 `NUMERIC` 或 `DOUBLE` 类型，存储枚举的索引（而不是名称）。

你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。 具体做法为：实现 `org.apache.ibatis.type.TypeHandler` 接口， 或继承一个很便利的类 `org.apache.ibatis.type.BaseTypeHandler` ， 然后可以选择性地将它映射到一个 JDBC 类型。比如：

```
// ExampleTypeHandler.java
@MappedJdbcTypes(JdbcType.VARCHAR)
public class ExampleTypeHandler extends BaseTypeHandler<String>
{

  @Override
  public void setNonNullParameter(PreparedStatement ps, int i, S
tring parameter, JdbcType jdbcType) throws SQLException {
    ps.setString(i, parameter);
  }

  @Override
  public String getNullableResult(ResultSet rs, String columnNam
e) throws SQLException {
    return rs.getString(columnName);
  }

  @Override
  public String getNullableResult(ResultSet rs, int columnIndex)
 throws SQLException {
    return rs.getString(columnIndex);
  }

  @Override
  public String getNullableResult(CallableStatement cs, int colu
mnIndex) throws SQLException {
    return cs.getString(columnIndex);
  }
}
```

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <typeHandler handler="org.mybatis.example.ExampleTypeHandler"/
>
</typeHandlers>
```

使用这个的类型处理器将会覆盖已经存在的处理 Java 的 String 类型属性和 VARCHAR 参数及结果的类型处理器。 要注意 MyBatis 不会窥探数据库元信息来决定使用哪种类型，所以你必须在参数和结果映射中指明那是 VARCHAR 类型的字段， 以使其能够绑定到正确的类型处理器上。 这是因为：MyBatis 直到语句被执行才清楚数据类型。

通过类型处理器的泛型，MyBatis 可以得知该类型处理器处理的 Java 类型，不过这种行为可以通过两种方法改变：

- 在类型处理器的配置元素（typeHandler element）上增加一个 `javaType` 属性（比如： `javaType="String"` ）；
- 在类型处理器的类上（TypeHandler class）增加一个 `@MappedTypes` 注解来

指定与其关联的 Java 类型列表。 如果在 `javaType` 属性中也同时指定，则注解方式将被忽略。

可以通过两种方式来指定被关联的 JDBC 类型：

- 在类型处理器的配置元素上增加一个 `javaType` 属性（比如：`javaType="VARCHAR"`）；
- 在类型处理器的类上（TypeHandler class）增加一个 `@MappedJdbcTypes` 注解来指定与其关联的 JDBC 类型列表。 如果在 `javaType` 属性中也同时指定，则注解方式将被忽略。

最后，可以让 MyBatis 为你查找类型处理器：

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <package name="org.mybatis.example"/>
</typeHandlers>
```

注意在使用自动检索（autodiscovery）功能的时候，只能通过注解方式来指定 JDBC 的类型。

你能创建一个泛型类型处理器，它可以处理多于一个类。为达到此目的， 需要增加一个接收该类作为参数的构造器，这样在构造一个类型处理器的时候 MyBatis 就会传入一个具体的类。

```
//GenericTypeHandler.java
public class GenericTypeHandler<E extends MyObject> extends Base
TypeHandler<E> {

  private Class<E> type;

  public GenericTypeHandler(Class<E> type) {
    if (type == null) throw new IllegalArgumentException("Type a
rgument cannot be null");
    this.type = type;
  }
  ...
```

`EnumTypeHandler` 和 `EnumOrdinalTypeHandler` 都是泛型类型处理器（generic TypeHandlers）， 我们将会在接下来的部分详细探讨。

处理枚举类型

若想映射枚举类型 `Enum`，则需要从 `EnumTypeHandler` 或者 `EnumOrdinalTypeHandler` 中选一个来使用。

比如说我们想存储取近似值时用到的舍入模式。默认情况下，MyBatis 会利用 `EnumTypeHandler` 来把 `Enum` 值转换成对应的名字。

注意 **`EnumTypeHandler`** 在某种意义上来说是比较特别的，其他的处理器只针对某个特定的类，而它不同，它会处理任意继承了 **`Enum`** 的类。

不过，我们可能不想存储名字，相反我们的 DBA 会坚持使用整形值代码。那也一样轻而易举： 在配置文件中把 `EnumOrdinalTypeHandler` 加到 `typeHandlers` 中即可， 这样每个 `RoundingMode` 将通过他们的序数值来映射成对应的整形。

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler" javaType="java.math.RoundingMode"/>
</typeHandlers>
```

但是怎样能将同样的 `Enum` 既映射成字符串又映射成整形呢？

自动映射器（auto-mapper）会自动地选用 `EnumOrdinalTypeHandler` 来处理，所以如果我们想用普通的 `EnumTypeHandler` ，就非要为那些 SQL 语句显式地设置要用到的类型处理器不可。

（下一节才开始讲映射器文件，所以如果是首次阅读该文档，你可能需要先越过这一步，过会再来看。）

```
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="org.apache.ibatis.submitted.rounding.Mapper">
    <resultMap type="org.apache.ibatis.submitted.rounding.User"
id="usermap">
        <id column="id" property="id"/>
        <result column="name" property="name"/>
        <result column="funkyNumber" property="funkyNumber"/>
        <result column="roundingMode" property="roundingMode"/>
    </resultMap>

    <select id="getUser" resultMap="usermap">
        select * from users
    </select>
    <insert id="insert">
        insert into users (id, name, funkyNumber, roundingMode)
values (
            #{id}, #{name}, #{funkyNumber}, #{roundingMode}
        )
    </insert>

    <resultMap type="org.apache.ibatis.submitted.rounding.User"
id="usermap2">
        <id column="id" property="id"/>
        <result column="name" property="name"/>
        <result column="funkyNumber" property="funkyNumber"/>
        <result column="roundingMode" property="roundingMode" ty
peHandler="org.apache.ibatis.type.EnumTypeHandler"/>
    </resultMap>
    <select id="getUser2" resultMap="usermap2">
        select * from users2
    </select>
    <insert id="insert2">
        insert into users2 (id, name, funkyNumber, roundingMode)
 values (
            #{id}, #{name}, #{funkyNumber}, #{roundingMode, type
Handler=org.apache.ibatis.type.EnumTypeHandler}
        )
    </insert>

</mapper>
```

注意，这里的 select 语句强制使用 `resultMap` 来代替 `resultType` 。

对象工厂（objectFactory）

MyBatis 每次创建结果对象的新实例时，它都会使用一个对象工厂（ObjectFactory）实例来完成。 默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。如果想覆盖对象工厂的默认行为，则可以通过创建自己的对象工厂来实现。比如：

```
// ExampleObjectFactory.java
public class ExampleObjectFactory extends DefaultObjectFactory {
  public Object create(Class type) {
    return super.create(type);
  }
  public Object create(Class type, List<Class> constructorArgTyp
es, List<Object> constructorArgs) {
    return super.create(type, constructorArgTypes, constructorAr
gs);
  }
  public void setProperties(Properties properties) {
    super.setProperties(properties);
  }
  public <T> boolean isCollection(Class<T> type) {
    return Collection.class.isAssignableFrom(type);
  }}
```

```
<!-- mybatis-config.xml -->
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
  <property name="someProperty" value="100"/>
</objectFactory>
```

ObjectFactory 接口很简单，它包含两个创建用的方法，一个是处理默认构造方法的，另外一个是处理带参数的构造方法的。 最后，setProperties 方法可以被用来配置 ObjectFactory，在初始化你的 ObjectFactory 实例后， objectFactory 元素体中定义的属性会被传递给 setProperties 方法。

插件（plugins）

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

- Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler (getParameterObject, setParameters)
- ResultSetHandler (handleResultSets, handleOutputParameters)
- StatementHandler (prepare, parameterize, batch, update, query)

这些类中方法的细节可以通过查看每个方法的签名来发现，或者直接查看 MyBatis 的发行包中的源代码。 假设你想做的不仅仅是监控方法的调用，那么你应该很好的了解正在重写的方法的行为。 因为如果在试图修改或重写已有方法的行为的时候，

你很可能在破坏 MyBatis 的核心模块。 这些都是更低层的类和方法，所以使用插件的时候要特别当心。

通过 MyBatis 提供的强大机制，使用插件是非常简单的，只需实现 Interceptor 接口，并指定了想要拦截的方法签名即可。

```
// ExamplePlugin.java
@Intercepts({@Signature(
  type= Executor.class,
  method = "update",
  args = {MappedStatement.class,Object.class})})
public class ExamplePlugin implements Interceptor {
  public Object intercept(Invocation invocation) throws Throwabl
e {
    return invocation.proceed();
  }
  public Object plugin(Object target) {
    return Plugin.wrap(target, this);
  }
  public void setProperties(Properties properties) {
  }
}
```

```
<!-- mybatis-config.xml -->
<plugins>
  <plugin interceptor="org.mybatis.example.ExamplePlugin">
    <property name="someProperty" value="100"/>
  </plugin>
</plugins>
```

上面的插件将会拦截在 Executor 实例中所有的 “update” 方法调用， 这里的 Executor 是负责执行低层映射语句的内部对象。

NOTE 覆盖配置类

除了用插件来修改 MyBatis 核心行为之外，还可以通过完全覆盖配置类来达到目的。只需继承后覆盖其中的每个方法，再把它传递到 sqlSessionFactoryBuilder.build(myConfig) 方法即可。再次重申，这可能会严重影响 MyBatis 的行为，务请慎之又慎。

配置环境（environments）

MyBatis 可以配置成适应多种环境，这种机制有助于将 SQL 映射应用于多种数据库之中， 现实情况下有多种理由需要这么做。例如，开发、测试和生产环境需要有不同的配置；或者共享相同 Schema 的多个生产数据库， 想使用相同的 SQL 映射。许多类似的用例。

不过要记住：尽管可以配置多个环境，每个 **SqlSessionFactory** 实例只能选择其一。

所以，如果你想连接两个数据库，就需要创建两个 SqlSessionFactory 实例，每个数据库对应一个。而如果是三个数据库，就需要三个实例，依此类推，记起来很简单：

- 每个数据库对应一个 **SqlSessionFactory** 实例

为了指定创建哪种环境，只要将它作为可选的参数传递给 SqlSessionFactoryBuilder 即可。可以接受环境配置的两个方法签名是：

```
SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reade
r, environment);
SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reade
r, environment,properties);
```

如果忽略了环境参数，那么默认环境将会被加载，如下所示：

```
SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reade
r);
SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reade
r,properties);
```

环境元素定义了如何配置环境。

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
      <property name="..." value="..."/>
    </transactionManager>
    <dataSource type="POOLED">
      <property name="driver" value="${driver}"/>
      <property name="url" value="${url}"/>
      <property name="username" value="${username}"/>
      <property name="password" value="${password}"/>
    </dataSource>
  </environment>
</environments>
```

注意这里的关键点:

- 默认的环境 ID（比如:default=”development”）。
- 每个 environment 元素定义的环境 ID（比如:id=”development”）。
- 事务管理器的配置（比如:type=”JDBC”）。
- 数据源的配置（比如:type=”POOLED”）。

默认的环境和环境 ID 是一目了然的。随你怎么命名，只要保证默认环境要匹配其中一个环境ID。

事务管理器（**transactionManager**）

在 MyBatis 中有两种类型的事务管理器（也就是 type=”[JDBC|MANAGED]”）：

- JDBC – 这个配置就是直接使用了 JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务范围。
- MANAGED – 这个配置几乎没做什么。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。 默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。例如:

```
&lt;transactionManager type="MANAGED"&gt;
  &lt;property name="closeConnection" value="false"/&gt;
&lt;/transactionManager&gt;
```

NOTE如果你正在使用 Spring + MyBatis，则没有必要配置事务管理器， 因为 Spring 模块会使用自带的管理器来覆盖前面的配置。

这两种事务管理器类型都不需要任何属性。它们不过是类型别名，换句话说，你可以使用 TransactionFactory 接口的实现类的完全限定名或类型别名代替它们。

```
public interface TransactionFactory {
  void setProperties(Properties props);
  Transaction newTransaction(Connection conn);
  Transaction newTransaction(DataSource dataSource, TransactionI
solationLevel level, boolean autoCommit);
}
```

任何在 XML 中配置的属性在实例化之后将会被传递给 setProperties() 方法。你也需要创建一个 Transaction 接口的实现类，这个接口也很简单：

```
public interface Transaction {
  Connection getConnection() throws SQLException;
  void commit() throws SQLException;
  void rollback() throws SQLException;
  void close() throws SQLException;
}
```

使用这两个接口，你可以完全自定义 MyBatis 对事务的处理。

数据源（**dataSource**）

dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。

- 许多 MyBatis 的应用程序将会按示例中的例子来配置数据源。然而它并不是必

须的。要知道为了方便使用延迟加载，数据源才是必须的。

有三种内建的数据源类型（也就是 type="[UNPOOLED|POOLED|JNDI]"）：

UNPOOLED– 这个数据源的实现只是每次被请求时打开和关闭连接。虽然一点慢，它对在及时可用连接方面没有性能要求的简单应用程序是一个很好的选择。 不同的数据库在这方面表现也是不一样的，所以对某些数据库来说使用连接池并不重要，这个配置也是理想的。UNPOOLED 类型的数据源仅仅需要配置以下 5 种属性：

- `driver` – 这是 JDBC 驱动的 Java 类的完全限定名（并不是JDBC驱动中可能包含的数据源类）。
- `url` – 这是数据库的 JDBC URL 地址。
- `username` – 登录数据库的用户名。
- `password` – 登录数据库的密码。
- `defaultTransactionIsolationLevel` – 默认的连接事务隔离级别。

作为可选项，你也可以传递属性给数据库驱动。要这样做，属性的前缀为“driver.”，例如：

- `driver.encoding=UTF8`

这将通过DriverManager.getConnection(url,driverProperties)方法传递值为 `UTF8` 的 `encoding` 属性给数据库驱动。

POOLED– 这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来，避免了创建新的连接实例时所必需的初始化和认证时间。 这是一种使得并发 Web 应用快速响应请求的流行处理方式。

除了上述提到 UNPOOLED 下的属性外，会有更多属性用来配置 POOLED 的数据源：

- `poolMaximumActiveConnections` – 在任意时间可以存在的活动（也就是正在使用）连接数量，默认值：10
- `poolMaximumIdleConnections` – 任意时间可能存在的空闲连接数。
- `poolMaximumCheckoutTime` – 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000 毫秒（即 20 秒）
- `poolTimeToWait` – 这是一个底层设置，如果获取连接花费的相当长的时间，它会给连接池打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直安静的失败），默认值：20000 毫秒（即 20 秒）。
- `poolPingQuery` – 发送到数据库的侦测查询，用来检验连接是否处在正常工作秩序中并准备接受请求。默认是“NO PING QUERY SET”，这会导致多数数据库驱动失败时带有一个恰当的错误消息。
- `poolPingEnabled` – 是否启用侦测查询。若开启，也必须使用一个可执行的 SQL 语句设置 `poolPingQuery` 属性（最好是一个非常快的 SQL），默认值：false。
- `poolPingConnectionsNotUsedFor` – 配置 poolPingQuery 的使用频度。这可以被设置成匹配具体的数据库连接超时时间，来避免不必要的侦测，默认值：0（即所有连接每一时刻都被侦测 — 当然仅当 poolPingEnabled 为 true 时适用）。

JNDI– 这个数据源的实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。这种数据源配置只需要两个属性：

- `initial_context` – 这个属性用来在 InitialContext 中寻找上下文（即，initialContext.lookup(initial_context)）。这是个可选属性，如果忽略，那么 data_source 属性将会直接从 InitialContext 中寻找。
- `data_source` – 这是引用数据源实例位置的上下文的路径。提供了 initial_context 配置时会在其返回的上下文中进行查找，没有提供时则直接在 InitialContext 中查找。

和其他数据源配置类似，可以通过添加前缀“env.”直接把属性传递给初始上下文。比如：

- `env.encoding=UTF8`

这就会在初始上下文（InitialContext）实例化时往它的构造方法传递值为 `UTF8` 的 `encoding` 属性。

通过需要实现接口 `org.apache.ibatis.datasource.DataSourceFactory` ，也可使用任何第三方数据源，：

```
public interface DataSourceFactory {
  void setProperties(Properties props);
  DataSource getDataSource();
}
```

`org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory` 可被用作父类来构建新的数据源适配器，比如下面这段插入 C3P0 数据源所必需的代码：

```
import org.apache.ibatis.datasource.unpooled.UnpooledDataSourceF
actory;
import com.mchange.v2.c3p0.ComboPooledDataSource;

public class C3P0DataSourceFactory extends UnpooledDataSourceFac
tory {

  public C3P0DataSourceFactory() {
    this.dataSource = new ComboPooledDataSource();
  }
}
```

为了令其工作，为每个需要 MyBatis 调用的 setter 方法中增加一个属性。下面是一个可以连接至 PostgreSQL 数据库的例子：

```
<dataSource type="org.myproject.C3P0DataSourceFactory">
  <property name="driver" value="org.postgresql.Driver"/>
  <property name="url" value="jdbc:postgresql:mydb"/>
  <property name="username" value="postgres"/>
  <property name="password" value="root"/>
</dataSource>
```

databaseIdProvider

MyBatis 可以根据不同的数据库厂商执行不同的语句，这种多厂商的支持是基于映射语句中的 `databaseId` 属性。MyBatis 会加载不带 `databaseId` 属性和带有匹配当前数据库 `databaseId` 属性的所有语句。如果同时找到带有 `databaseId` 和不带 `databaseId` 的相同语句，则后者会被舍弃。为支持多厂商特性只要像下面这样在 mybatis-config.xml 文件中加入 `databaseIdProvider` 即可：

```
<databaseIdProvider type="DB_VENDOR" />
```

这里的 DB_VENDOR 会通过 `DatabaseMetaData#getDatabaseProductName()` 返回的字符串进行设置。由于通常情况下这个字符串都非常长而且相同产品的不同版本会返回不同的值，所以最好通过设置属性别名来使其变短，如下：

```
<databaseIdProvider type="DB_VENDOR">
  <property name="SQL Server" value="sqlserver"/>
  <property name="DB2" value="db2"/>
  <property name="Oracle" value="oracle" />
</databaseIdProvider>
```

在有 properties 时，DB_VENDOR databaseIdProvider 的将被设置为第一个能匹配数据库产品名称的属性键对应的值，如果没有匹配的属性将会设置为 “null”。在这个例子中，如果 `getDatabaseProductName()` 返回“Oracle (DataDirect)”，databaseId 将被设置为“oracle”。

你可以通过实现接口 `org.apache.ibatis.mapping.DatabaseIdProvider` 并在 mybatis-config.xml 中注册来构建自己的 DatabaseIdProvider：

```
public interface DatabaseIdProvider {
  void setProperties(Properties p);
  String getDatabaseId(DataSource dataSource) throws SQLExceptio
n;
}
```

映射器（mappers）

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要定义 SQL 映射语句了。但是首先我们需要告诉 MyBatis 到哪里去找到这些语句。 Java 在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。你可以使用相对于类路径的资源引用， 或完全限定资源定位符（包括 `file:///` 的 URL），或类名和包名等。例如：

```
<!-- Using classpath relative resources -->
<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>
```

```
<!-- Using url fully qualified paths -->
<mappers>
  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>
```

```
<!-- Using mapper interface classes -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>
```

```
<!-- Register all interfaces in a package as mappers -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

这些配置会告诉了 MyBatis 去哪里找映射文件，剩下的细节就应该是每个 SQL 映射文件了，也就是接下来我们要讨论的。

Mapper XML 文件

MyBatis 的真正强大在于它的映射语句，也是它的魔力所在。由于它的异常强大，映射器的 XML 文件就显得相对简单。如果拿它跟具有相同功能的 JDBC 代码进行对比，你会立即发现省掉了将近 95% 的代码。MyBatis 就是针对 SQL 构建的，并且比普通的方法做的更好。

SQL 映射文件有很少的几个顶级元素（按照它们应该被定义的顺序）：

- `cache` – 给定命名空间的缓存配置。
- `cache-ref` – 其他命名空间缓存配置的引用。
- `resultMap` – 是最复杂也是最强大的元素，用来描述如何从数据库结果集中来加载对象。
- `~~parameterMap~~` ~~– 已废弃！老式风格的参数映射。内联参数是首选,这个元素可能在将来被移除，这里不会记录。~~
- `sql` – 可被其他语句引用的可重用语句块。
- `insert` – 映射插入语句
- `update` – 映射更新语句
- `delete` – 映射删除语句
- `select` – 映射查询语句

下一部分将从语句本身开始来描述每个元素的细节。

select

查询语句是 MyBatis 中最常用的元素之一，光能把数据存到数据库中价值并不大，如果还能重新取出来才有用，多数应用也都是查询比修改要频繁。对每个插入、更新或删除操作，通常对应多个查询操作。这是 MyBatis 的基本原则之一，也是将焦点和努力放到查询和结果映射的原因。简单查询的 select 元素是非常简单的。比如：

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

这个语句被称作 selectPerson，接受一个 int（或 Integer）类型的参数，并返回一个 HashMap 类型的对象，其中的键是列名，值便是结果行中的对应值。

注意参数符号：

```
#{id}
```

这就告诉 MyBatis 创建一个预处理语句参数，通过 JDBC，这样的一个参数在 SQL 中会由一个“?”来标识，并被传递到一个新的预处理语句中，就像这样：

```
// Similar JDBC code, NOT MyBatis…
String selectPerson = "SELECT * FROM PERSON WHERE ID=?";
PreparedStatement ps = conn.prepareStatement(selectPerson);
ps.setInt(1,id);
```

当然，这需要很多单独的 JDBC 的代码来提取结果并将它们映射到对象实例中，这就是 MyBatis 节省你时间的地方。我们需要深入了解参数和结果映射，细节部分我们下面来了解。

select 元素有很多属性允许你配置，来决定每条语句的作用细节。

```
<select
  id="selectPerson"
  parameterType="int"
  parameterMap="deprecated"
  resultType="hashmap"
  resultMap="personResultMap"
  flushCache="false"
  useCache="true"
  timeout="10000"
  fetchSize="256"
  statementType="PREPARED"
  resultSetType="FORWARD_ONLY">
```

Select Attributes

属性	描述
`id`	在命名空间中唯一的标识符，可以被用来引用这条语句。
`parameterType`	将会传入这条语句的参数类的完全限定名或别名。这个属性是可选的，因为 MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数，默认值为 unset。
~~parameterMap~~	~~这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterType 属性。~~
`resultType`	从这条语句中返回的期望类型的类的完全限定名或别名。注意如果是集合情形，那应该是集合可以包含的类型，而不能是集合本身。使用 resultType 或 resultMap，但不能同时使用。
`resultMap`	外部 resultMap 的命名引用。结果集的映射是 MyBatis 最强大的特性，对其有一个很好的理解的话，许多复杂映射的情形都能迎刃而解。使用 resultMap 或 resultType，但不能同时使用。

`flushCache`	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空，默认值：false。
`useCache`	将其设置为 true，将会导致本条语句的结果被二级缓存，默认值：对 select 元素为 true。
`timeout`	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为 unset（依赖驱动）。
`fetchSize`	这是尝试影响驱动程序每次批量返回的结果行数和这个设置值相等。默认值为 unset（依赖驱动）。
`statementType`	STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
`resultSetType`	FORWARD_ONLY，SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE 中的一个，默认值为 unset （依赖驱动）。
`databaseId`	如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 databaseId 的语句；如果带或者不带的语句都有，则不带的会被忽略。
`resultOrdered`	这个设置仅针对嵌套结果 select 语句适用：如果为 true，就是假设包含了嵌套结果集或是分组了，这样的话当返回一个主结果行的时候，就不会发生有对前面结果集的引用的情况。这就使得在获取嵌套的结果集的时候不至于导致内存不够用。默认值： `false` 。
`resultSets`	这个设置仅对多结果集的情况适用，它将列出语句执行后返回的结果集并每个结果集给一个名称，名称是逗号分隔的。

insert, update 和 delete

数据变更语句 insert，update 和 delete 的实现非常接近：

```
<insert
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  keyProperty=""
  keyColumn=""
  useGeneratedKeys=""
  timeout="20">

<update
  id="updateAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">

<delete
  id="deleteAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">
```

Insert, Update 和 Delete 的属性

属性	描述
`id`	命名空间中的唯一标识符，可被用来代表这条语句。
`parameterType`	将要传入语句的参数的完全限定类名或别名。这个属性是可选的，因为 MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数，默认值为 unset。
`~~parameterMap~~`	~~这是引用外部 parameterMap 的已经被废弃的方法。使用内联参数映射和 parameterType 属性。~~
`flushCache`	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空，默认值：true（对应插入、更新和删除语句）。
`timeout`	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为 unset（依赖驱动）。
`statementType`	STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
`useGeneratedKeys`	（仅对 insert 和 update 有用）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键（比如：像 MySQL 和 SQL Server 这样的关系数据库管理系统的自动递增字段），默认值：false。
`keyProperty`	（仅对 insert 和 update 有用）唯一标记一个属性，MyBatis 会通过 getGeneratedKeys 的返回值或者通过 insert 语句的 selectKey 子元素设置它的键值，默认： `unset` 。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。
`keyColumn`	（仅对 insert 和 update 有用）通过生成的键值设置表中的列名，这个设置仅在某些数据库（像 PostgreSQL）是必须的，当主键列不是表中的第一列的时候需要设置。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。
`databaseId`	如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 databaseId 的语句；如果带或者不带的语句都有，则不带的会被忽略。

下面就是 insert，update 和 delete 语句的示例：

```
<insert id="insertAuthor">
  insert into Author (id,username,password,email,bio)
  values (#{id},#{username},#{password},#{email},#{bio})
</insert>

<update id="updateAuthor">
  update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
  where id = #{id}
</update>

<delete id="deleteAuthor">
  delete from Author where id = #{id}
</delete>
```

如前所述，插入语句的配置规则更加丰富，在插入语句里面有一些额外的属性和子元素用来处理主键的生成，而且有多种生成方式。

首先，如果你的数据库支持自动生成主键的字段（比如 MySQL 和 SQL Server），那么你可以设置 useGeneratedKeys=”true”，然后再把 keyProperty 设置到目标属性上就OK了。例如，如果上面的 Author 表已经对 id 使用了自动生成的列类型，那么语句可以修改为:

```
<insert id="insertAuthor" useGeneratedKeys="true"
    keyProperty="id">
  insert into Author (username,password,email,bio)
  values (#{username},#{password},#{email},#{bio})
</insert>
```

对于不支持自动生成类型的数据库或可能不支持自动生成主键 JDBC 驱动来说，MyBatis 有另外一种方法来生成主键。

这里有一个简单（甚至很傻）的示例，它可以生成一个随机 ID（你最好不要这么做，但这里展示了 MyBatis 处理问题的灵活性及其所关心的广度）：

```
<insert id="insertAuthor">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDU
MMY1
  </selectKey>
  insert into Author
    (id, username, password, email,bio, favourite_section)
  values
    (#{id}, #{username}, #{password}, #{email}, #{bio}, #{favour
iteSection,jdbcType=VARCHAR})
</insert>
```

在上面的示例中，selectKey 元素将会首先运行，Author 的 id 会被设置，然后插入语句会被调用。这给你了一个和数据库中来处理自动生成的主键类似的行为，避免了使 Java 代码变得复杂。

selectKey 元素描述如下：

```
<selectKey
  keyProperty="id"
  resultType="int"
  order="BEFORE"
  statementType="PREPARED">
```

selectKey 的属性

属性	描述
`keyProperty`	selectKey 语句结果应该被设置的目标属性。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。
`keyColumn`	匹配属性的返回结果集中的列名称。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。
`resultType`	结果的类型。MyBatis 通常可以推算出来，但是为了更加确定写上也不会有什么问题。MyBatis 允许任何简单类型用作主键的类型，包括字符串。如果希望作用于多个生成的列，则可以使用一个包含期望属性的 Object 或一个 Map。
`order`	这可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它会首先选择主键，设置 keyProperty 然后执行插入语句。如果设置为 AFTER，那么先执行插入语句，然后是 selectKey 元素 - 这和像 Oracle 的数据库相似，在插入语句内部可能有嵌入索引调用。
`statementType`	与前面相同，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE 语句的映射类型，分别代表 PreparedStatement 和 CallableStatement 类型。

sql

这个元素可以被用来定义可重用的 SQL 代码段，可以包含在其他语句中。它可以被静态地(在加载参数) 参数化. 不同的属性值通过包含的实例变化. 比如：

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.pa
ssword </sql>
```

这个 SQL 片段可以被包含在其他语句中，例如：

```
<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias" value="t
1"/></include>,
    <include refid="userColumns"><property name="alias" value="t
2"/></include>
  from some_table t1
    cross join some_table t2
</select>
```

属性值可以用于包含的refid属性或者包含的字句里面的属性值，例如：

```
<sql id="sometable">
  ${prefix}Table
</sql>

<sql id="someinclude">
  from
    <include refid="${include_target}"/>
</sql>

<select id="select" resultType="map">
  select
    field1, field2, field3
  <include refid="someinclude">
    <property name="prefix" value="Some"/>
    <property name="include_target" value="sometable"/>
  </include>
</select>
```

参数（Parameters）

前面的所有语句中你所见到的都是简单参数的例子，实际上参数是 MyBatis 非常强大的元素，对于简单的做法，大概 90% 的情况参数都很少，比如：

```
<select id="selectUsers" resultType="User">
  select id, username, password
  from users
  where id = #{id}
</select>
```

上面的这个示例说明了一个非常简单的命名参数映射。参数类型被设置为 `int`，这样这个参数就可以被设置成任何内容。原生的类型或简单数据类型（比如整型和字符串）因为没有相关属性，它会完全用参数值来替代。然而，如果传入一个复杂的对象，行为就会有一点不同了。比如：

```
<insert id="insertUser" parameterType="User">
  insert into users (id, username, password)
  values (#{id}, #{username}, #{password})
</insert>
```

如果 User 类型的参数对象传递到了语句中，id、username 和 password 属性将会被查找，然后将它们的值传入预处理语句的参数中。

这点对于向语句中传参是比较好的而且又简单，不过参数映射的功能远不止于此。

首先，像 MyBatis 的其他部分一样，参数也可以指定一个特殊的数据类型。

```
#{property,javaType=int,jdbcType=NUMERIC}
```

像 MyBatis 的剩余部分一样，javaType 通常可以从参数对象中来去确定，前提是只要对象不是一个 HashMap。那么 javaType 应该被确定来保证使用正确类型处理器。

NOTE 如果 null 被当作值来传递，对于所有可能为空的列，JDBC Type 是需要的。你可以自己通过阅读预处理语句的 setNull() 方法的 JavaDocs 文档来研究这种情况。

为了以后定制类型处理方式，你也可以指定一个特殊的类型处理器类（或别名），比如：

```
#{age,javaType=int,jdbcType=NUMERIC,typeHandler=MyTypeHandler}
```

尽管看起来配置变得越来越繁琐，但实际上是很少去设置它们。

对于数值类型，还有一个小数保留位数的设置，来确定小数点后保留的位数。

```
#{height,javaType=double,jdbcType=NUMERIC,numericScale=2}
```

最后，mode 属性允许你指定 IN，OUT 或 INOUT 参数。如果参数为 OUT 或 INOUT，参数对象属性的真实值将会被改变，就像你在获取输出参数时所期望的那样。如果 mode 为 OUT（或 INOUT），而且 jdbcType 为 CURSOR(也就是 Oracle 的 REFCURSOR)，你必须指定一个 resultMap 来映射结果集到参数类型。要注意这里的 javaType 属性是可选的，如果左边的空白是 jdbcType 的 CURSOR 类型，它会自动地被设置为结果集。

```
#{department, mode=OUT, jdbcType=CURSOR, javaType=ResultSet, res
ultMap=departmentResultMap}
```

MyBatis 也支持很多高级的数据类型，比如结构体，但是当注册 out 参数时你必须告诉它语句类型名称。比如（再次提示，在实际中要像这样不能换行）：

```
#{middleInitial, mode=OUT, jdbcType=STRUCT, jdbcTypeName=MY_TYPE
, resultMap=departmentResultMap}
```

尽管所有这些强大的选项很多时候你只简单指定属性名，其他的事情 MyBatis 会自己去推断，最多你需要为可能为空的列名指定 `jdbcType` 。

```
#{firstName}
#{middleInitial,jdbcType=VARCHAR}
#{lastName}
```

字符串替换

默认情况下,使用#{}格式的语法会导致 MyBatis 创建预处理语句属性并安全地设置值（比如?）。这样做更安全，更迅速，通常也是首选做法，不过有时你只是想直接在 SQL 语句中插入一个不改变的字符串。比如，像 ORDER BY，你可以这样来使用：

```
ORDER BY ${columnName}
```

这里 MyBatis 不会修改或转义字符串。

NOTE 以这种方式接受从用户输出的内容并提供给语句中不变的字符串是不安全的，会导致潜在的 SQL 注入攻击，因此要么不允许用户输入这些字段，要么自行转义并检验。

Result Maps

resultMap 元素是 MyBatis 中最重要最强大的元素。它就是让你远离 90%的需要从结果 集中取出数据的 JDBC 代码的那个东西, 而且在一些情形下允许你做一些 JDBC 不支持的事 情。 事实上, 编写相似于对复杂语句联合映射这些等同的代码,

也许可以跨过上千行的代码。 ResultMap 的设计就是简单语句不需要明确的结果映射,而很多复杂语句确实需要描述它们 的关系。

你已经看到简单映射语句的示例了,但没有明确的 resultMap。比如:

```
<select id="selectUsers" resultType="map">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

这样一个语句简单作用于所有列被自动映射到 HashMap 的键上,这由 resultType 属性 指定。这在很多情况下是有用的,但是 HashMap 不能很好描述一个领域模型。那样你的应 用程序将会使用 JavaBeans 或 POJOs(Plain Old Java Objects,普通 Java 对象)来作为领域 模型。MyBatis 对两者都支持。看看下面这个 JavaBean:

```
package com.someapp.model;
public class User {
  private int id;
  private String username;
  private String hashedPassword;

  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getUsername() {
    return username;
  }
  public void setUsername(String username) {
    this.username = username;
  }
  public String getHashedPassword() {
    return hashedPassword;
  }
  public void setHashedPassword(String hashedPassword) {
    this.hashedPassword = hashedPassword;
  }
}
```

基于 JavaBean 的规范,上面这个类有 3 个属性:id,username 和 hashedPassword。这些 在 select 语句中会精确匹配到列名。

这样的一个 JavaBean 可以被映射到结果集,就像映射到 HashMap 一样简单。

```
<select id="selectUsers" resultType="com.someapp.model.User">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

要记住类型别名是你的伙伴。使用它们你可以不用输入类的全路径。比如:

```
<!-- In mybatis-config.xml file -->
<typeAlias type="com.someapp.model.User" alias="User"/>

<!-- In SQL Mapping XML file -->
<select id="selectUsers" resultType="User">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

这些情况下,MyBatis 会在幕后自动创建一个 ResultMap,基于属性名来映射列到 JavaBean 的属性上。如果列名没有精确匹配,你可以在列名上使用 select 字句的别名(一个 基本的 SQL 特性)来匹配标签。比如:

```
<select id="selectUsers" resultType="User">
  select
    user_id             as "id",
    user_name           as "userName",
    hashed_password     as "hashedPassword"
  from some_table
  where id = #{id}
</select>
```

ResultMap 最优秀的地方你已经了解了很多了,但是你还没有真正的看到一个。这些简 单的示例不需要比你看到的更多东西。 只是出于示例的原因, 让我们来看看最后一个示例中 外部的 resultMap 是什么样子的,这也是解决列名不匹配的另外一种方式。

```
<resultMap id="userResultMap" type="User">
  <id property="id" column="user_id" />
  <result property="username" column="user_name"/>
  <result property="password" column="hashed_password"/>
</resultMap>
```

引用它的语句使用 resultMap 属性就行了(注意我们去掉了 resultType 属性)。比如:

```
<select id="selectUsers" resultMap="userResultMap">
  select user_id, user_name, hashed_password
  from some_table
  where id = #{id}
</select>
```

如果世界总是这么简单就好了。

高级结果映射

MyBatis 创建的一个想法:数据库不用永远是你想要的或需要它们是什么样的。而我们 最喜欢的数据库最好是第三范式或 BCNF 模式,但它们有时不是。如果可能有一个单独的 数据库映射,所有应用程序都可以使用它,这是非常好的,但有时也不是。结果映射就是 MyBatis 提供处理这个问题的答案。

比如,我们如何映射下面这个语句?

```
<!-- Very Complex Statement -->
<select id="selectBlogDetails" resultMap="detailedBlogResultMap"
>
  select
       B.id as blog_id,
       B.title as blog_title,
       B.author_id as blog_author_id,
       A.id as author_id,
       A.username as author_username,
       A.password as author_password,
       A.email as author_email,
       A.bio as author_bio,
       A.favourite_section as author_favourite_section,
       P.id as post_id,
       P.blog_id as post_blog_id,
       P.author_id as post_author_id,
       P.created_on as post_created_on,
       P.section as post_section,
       P.subject as post_subject,
       P.draft as draft,
       P.body as post_body,
       C.id as comment_id,
       C.post_id as comment_post_id,
       C.name as comment_name,
       C.comment as comment_text,
       T.id as tag_id,
       T.name as tag_name
  from Blog B
       left outer join Author A on B.author_id = A.id
       left outer join Post P on B.id = P.blog_id
       left outer join Comment C on P.id = C.post_id
       left outer join Post_Tag PT on PT.post_id = P.id
       left outer join Tag T on PT.tag_id = T.id
  where B.id = #{id}
</select>
```

你可能想把它映射到一个智能的对象模型,包含一个作者写的博客,有很多的博文,每篇博文有零条或多条的评论和标签。 下面是一个完整的复杂结果映射例子 (假设作者, 博客, 博文, 评论和标签都是类型的别名) 我们来看看, 。 但是不用紧张, 我们会一步一步来说明。 当天最初它看起来令人生畏,但实际上非常简单。

```
<!-- Very Complex Result Map -->
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
    <result property="favouriteSection" column="author_favourite
_section"/>
  </association>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <association property="author" javaType="Author"/>
    <collection property="comments" ofType="Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" ofType="Tag" >
      <id property="id" column="tag_id"/>
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost"/>
    </discriminator>
  </collection>
</resultMap>
```

resultMap 元素有很多子元素和一个值得讨论的结构。 下面是 resultMap 元素的概念视图

resultMap

- `constructor` - 类在实例化时,用来注入结果到构造方法中
  - `idArg` - ID 参数;标记结果作为 ID 可以帮助提高整体效能
  - `arg` - 注入到构造方法的一个普通结果
- `id` – 一个 ID 结果;标记结果作为 ID 可以帮助提高整体效能
- `result` – 注入到字段或 JavaBean 属性的普通结果
- `association` – 一个复杂的类型关联;许多结果将包成这种类型
  - 嵌入结果映射 – 结果映射自身的关联,或者参考一个
- `collection` – 复杂类型的集
  - 嵌入结果映射 – 结果映射自身的集,或者参考一个
- `discriminator` – 使用结果值来决定使用哪个结果映射
  - `case` – 基于某些值的结果映射
    - 嵌入结果映射 – 这种情形结果也映射它本身,因此可以包含很多相 同

的元素,或者它可以参照一个外部的结果映射。

ResultMap Attributes

Attribute	Description
`id`	A unique identifier in this namespace that can be used to reference this result map.
`type`	A fully qualified Java class name, or a type alias (see the table above for the list of built-in type aliases).
`autoMapping`	If present, MyBatis will enable or disable the automapping for this ResultMap. This attribute overrides the global autoMappingBehavior. Default: unset.

最佳实践 通常逐步建立结果映射。单元测试的真正帮助在这里。如果你尝试创建一次创建一个向上面示例那样的巨大的结果映射, 那么可能会有错误而且很难去控制它 来工作。开始简单一些,一步一步的发展。而且要进行单元测试!使用该框架的缺点是 它们有时是黑盒(是否可见源代码) 。你确定你实现想要的行为的最好选择是编写单元 测试。它也可以你帮助得到提交时的错误。

下面一部分将详细说明每个元素。

id & result

```
<id property="id" column="post_id"/>
<result property="subject" column="post_subject"/>
```

这些是结果映射最基本内容。id 和 result 都映射一个单独列的值到简单数据类型(字符 串,整型,双精度浮点数,日期等)的单独属性或字段。

这两者之间的唯一不同是 id 表示的结果将是当比较对象实例时用到的标识属性。这帮 助来改进整体表现,特别是缓存和嵌入结果映射(也就是联合映射) 。

每个都有一些属性:

Id and Result Attributes

属性	描述
`property`	映射到列结果的字段或属性。如果匹配的是存在的,和给定名称相同 的 JavaBeans 的属性,那么就会使用 。否则 MyBatis 将会寻找给定名称 property 的字段。这两种情形你可以使用通常点式的复杂属性导航。比如,你 可以这样映射一些东西: “username” ,或者映射到一些复杂的东西: “address.street.number” 。
`column`	从数据库中得到的列名,或者是列名的重命名标签。这也是通常和会 传递给 resultSet.getString(columnName)方法参数中相同的字符串。
`javaType`	一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名 的列表) 。如果你映射到一个 JavaBean,MyBatis 通常可以断定类型。 然而,如果你映射到的是 HashMap,那么你应该明确地指定 javaType 来保证所需的行为。
`jdbcType`	在这个表格之后的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅 仅需要对插入,更新和删除操作可能为空的列进行处理。这是 JDBC jdbcType 的需要,而不是 MyBatis 的。如果你直接使用 JDBC 编程,你需要指定 这个类型-但仅仅对可能为空的值。
`typeHandler`	我们在前面讨论过默认的类型处理器。使用这个属性,你可以覆盖默 认的类型处理器。这个属性值是类的完全限定名或者是一个类型处理 器的实现,或者是类型别名。

支持的 JDBC 类型

为了未来的参考,MyBatis 通过包含的 jdbcType 枚举型,支持下面的 JDBC 类型。

`BIT`	`FLOAT`	`CHAR`	`TIMESTAMP`	`OTHER`
`TINYINT`	`REAL`	`VARCHAR`	`BINARY`	`BLOG`
SMALLINT	DOUBLE	LONGVARCHAR	VARBINARY	CLOB
`INTEGER`	`NUMERIC`	`DATE`	`LONGVARBINARY`	`BOOLEAN`
BIGINT	DECIMAL	TIME	NULL	CURSOR

构造方法

```
<constructor>
   <idArg column="id" javaType="int"/>
   <arg column="username" javaType="String"/>
</constructor>
```

对于大多数数据传输对象(Data Transfer Object,DTO)类型,属性可以起作用,而且像你绝大多数的领域模型, 指令也许是你想使用一成不变的类的地方。 通常包含引用或查询数 据的表很少或基本不变的话对一成不变的类来说是合适的。 构造方法注入允许你在初始化时 为类设置属性的值,而不用暴露出公有方法。MyBatis 也支持私有属性和私有 JavaBeans 属 性来达到这个目的,但是一些人更青睐构造方法注入。构造方法元素支持这个。

看看下面这个构造方法:

```
public class User {
   //...
   public User(int id, String username) {
     //...
  }
//...
}
```

为了向这个构造方法中注入结果,MyBatis 需要通过它的参数的类型来标识构造方法。 Java 没有自查(反射)参数名的方法。所以当创建一个构造方法元素时,保证参数是按顺序 排列的,而且数据类型也是确定的。

```
<constructor>
   <idArg column="id" javaType="int"/>
   <arg column="username" javaType="String"/>
</constructor>
```

剩余的属性和规则和固定的 id 和 result 元素是相同的。

属性	描述
`column`	来自数据库的类名,或重命名的列标签。这和通常传递给 resultSet.getString(columnName)方法的字符串是相同的。
`javaType`	一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列表)。 如果你映射到一个 JavaBean,MyBatis 通常可以断定类型。然而,如 果你映射到的是 HashMap,那么你应该明确地指定 javaType 来保证所需的 行为。
`jdbcType`	在这个表格之前的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅仅 需要对插入, 更新和删除操作可能为空的列进行处理。这是 JDBC 的需要, jdbcType 而不是 MyBatis 的。如果你直接使用 JDBC 编程,你需要指定这个类型-但 仅仅对可能为空的值。
`typeHandler`	我们在前面讨论过默认的类型处理器。使用这个属性,你可以覆盖默认的 类型处理器。 这个属性值是类的完全限定名或者是一个类型处理器的实现, 或者是类型别名。
`select`	The ID of another mapped statement that will load the complex type required by this property mapping. The values retrieved from columns specified in the column attribute will be passed to the target select statement as parameters. See the Association element for more.
`resultMap`	This is the ID of a ResultMap that can map the nested results of this argument into an appropriate object graph. This is an alternative to using a call to another select statement. It allows you to join multiple tables together into a single `ResultSet` . Such a `ResultSet` will contain duplicated, repeating groups of data that needs to be decomposed and mapped properly to a nested object graph. To facilitate this, MyBatis lets you "chain" result maps together, to deal with the nested results. See the Association element below for more.

`关联`

```
<association property="author" column="blog_author_id" javaType=
"Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
</association>
```

关联元素处理“有一个”类型的关系。比如,在我们的示例中,一个博客有一个用户。关联映射就工作于这种结果之上。你指定了目标属性,来获取值的列,属性的 java 类型(很 多情况下 MyBatis 可以自己算出来) ,如果需要的话还有 jdbc 类型,如果你想覆

盖或获取的 结果值还需要类型控制器。

关联中不同的是你需要告诉 MyBatis 如何加载关联。MyBatis 在这方面会有两种不同的 方式:

- 嵌套查询:通过执行另外一个 SQL 映射语句来返回预期的复杂类型。
- 嵌套结果:使用嵌套结果映射来处理重复的联合结果的子集。首先,然让我们来查看这个元素的属性。所有的你都会看到,它和普通的只由 select 和

resultMap 属性的结果映射不同。

属性	描述
`property`	映射到列结果的字段或属性。如果匹配的是存在的,和给定名称相同的 property JavaBeans 的属性, 那么就会使用。 否则 MyBatis 将会寻找给定名称的字段。 这两种情形你可以使用通常点式的复杂属性导航。比如,你可以这样映射 一 些 东 西 :“ username ”, 或 者 映 射 到 一 些 复 杂 的 东 西 : “address.street.number” 。
`javaType`	一个 Java 类的完全限定名,或一个类型别名(参考上面内建类型别名的列 表) 。如果你映射到一个 JavaBean,MyBatis 通常可以断定类型。然而,如 javaType 果你映射到的是 HashMap,那么你应该明确地指定 javaType 来保证所需的 行为。
`jdbcType`	在这个表格之前的所支持的 JDBC 类型列表中的类型。JDBC 类型是仅仅 需要对插入, 更新和删除操作可能为空的列进行处理。这是 JDBC 的需要, jdbcType 而不是 MyBatis 的。如果你直接使用 JDBC 编程,你需要指定这个类型-但 仅仅对可能为空的值。
`typeHandler`	我们在前面讨论过默认的类型处理器。使用这个属性,你可以覆盖默认的 typeHandler 类型处理器。 这个属性值是类的完全限定名或者是一个类型处理器的实现, 或者是类型别名。

关联的嵌套查询

属性	描述
`column`	来自数据库的类名,或重命名的列标签。这和通常传递给 resultSet.getString(columnName)方法的字符串是相同的。 column 注 意 : 要 处 理 复 合 主 键 , 你 可 以 指 定 多 个 列 名 通 过 column= ” {prop1=col1,prop2=col2} ” 这种语法来传递给嵌套查询语 句。这会引起 prop1 和 prop2 以参数对象形式来设置给目标嵌套查询语句。
`select`	另外一个映射语句的 ID,可以加载这个属性映射需要的复杂类型。获取的 在列属性中指定的列的值将被传递给目标 select 语句作为参数。表格后面 有一个详细的示例。 select 注 意 : 要 处 理 复 合 主 键 , 你 可 以 指 定 多 个 列 名 通 过 column= ” {prop1=col1,prop2=col2} ” 这种语法来传递给嵌套查询语 句。这会引起 prop1 和 prop2 以参数对象形式来设置给目标嵌套查询语句。
`fetchType`	Optional. Valid values are `lazy` and `eager`. If present, it supersedes the global configuration parameter `lazyLoadingEnabled` for this mapping.

示例:

```
<resultMap id="blogResult" type="Blog">
  <association property="author" column="author_id" javaType="Au
thor" select="selectAuthor"/>
</resultMap>

<select id="selectBlog" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectAuthor" resultType="Author">
  SELECT * FROM AUTHOR WHERE ID = #{id}
</select>
```

我们有两个查询语句:一个来加载博客,另外一个来加载作者,而且博客的结果映射描述了“selectAuthor”语句应该被用来加载它的 author 属性。

其他所有的属性将会被自动加载,假设它们的列和属性名相匹配。

这种方式很简单, 但是对于大型数据集合和列表将不会表现很好。 问题就是我们熟知的 “N+1 查询问题”。概括地讲,N+1 查询问题可以是这样引起的:

- 你执行了一个单独的 SQL 语句来获取结果列表(就是“+1”)。
- 对返回的每条记录,你执行了一个查询语句来为每个加载细节(就是“N”)。

这个问题会导致成百上千的 SQL 语句被执行。这通常不是期望的。

MyBatis 能延迟加载这样的查询就是一个好处,因此你可以分散这些语句同时运行的消 耗。然而,如果你加载一个列表,之后迅速迭代来访问嵌套的数据,你会调用所有的延迟加 载,这样的行为可能是很糟糕的。

所以还有另外一种方法。

关联的嵌套结果

属性	描述
`resultMap`	这是结果映射的 ID,可以映射关联的嵌套结果到一个合适的对象图中。这 是一种替代方法来调用另外一个查询语句。这允许你联合多个表来合成到 resultMap 一个单独的结果集。这样的结果集可能包含重复,数据的重复组需要被分 解,合理映射到一个嵌套的对象图。为了使它变得容易,MyBatis 让你“链 接”结果映射,来处理嵌套结果。一个例子会很容易来仿照,这个表格后 面也有一个示例。
`columnPrefix`	When joining multiple tables, you would have to use column alias to avoid duplicated column names in the ResultSet. Specifying columnPrefix allows you to map such columns to an external resultMap. Please see the example explained later in this section.
`notNullColumn`	By default a child object is created only if at least one of the columns mapped to the child's properties is non null. With this attribute you can change this behaviour by specifiying which columns must have a value so MyBatis will create a child object only if any of those columns is not null. Multiple column names can be specified using a comma as a separator. Default value: unset.
`autoMapping`	If present, MyBatis will enable or disable auto-mapping when mapping the result to this property. This attribute overrides the global autoMappingBehavior. Note that it has no effect on an external resultMap, so it is pointless to use it with `select` or `resultMap` attribute. Default value: unset.

在上面你已经看到了一个非常复杂的嵌套关联的示例。 下面这个是一个非常简单的示例 来说明它如何工作。代替了执行一个分离的语句,我们联合博客表和作者表在一起,就像:

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id            as blog_id,
    B.title         as blog_title,
    B.author_id     as blog_author_id,
    A.id            as author_id,
    A.username      as author_username,
    A.password      as author_password,
    A.email         as author_email,
    A.bio           as author_bio
  from Blog B left outer join Author A on B.author_id = A.id
  where B.id = #{id}
</select>
```

注意这个联合查询, 以及采取保护来确保所有结果被唯一而且清晰的名字来重命名。 这使得映射非常简单。现在我们可以映射这个结果:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaTyp
e="Author" resultMap="authorResult"/>
</resultMap>

<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>
```

在上面的示例中你可以看到博客的作者关联代表着“authorResult”结果映射来加载作者实例。

非常重要: 在嵌套据诶过映射中 id 元素扮演了非常重要的角色。应应该通常指定一个 或多个属性,它们可以用来唯一标识结果。实际上就是如果你离开她了,但是有一个严重的 性能问题时 MyBatis 仍然可以工作。选择的属性越少越好,它们可以唯一地标识结果。主键 就是一个显而易见的选择(尽管是联合主键)。

现在,上面的示例用了外部的结果映射元素来映射关联。这使得 Author 结果映射可以 重用。然而,如果你不需要重用它的话,或者你仅仅引用你所有的结果映射合到一个单独描 述的结果映射中。你可以嵌套结果映射。这里给出使用这种方式的相同示例:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
  </association>
</resultMap>
```

What if the blog has a co-author? The select statement would look like:

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id            as blog_id,
    B.title         as blog_title,
    A.id            as author_id,
    A.username      as author_username,
    A.password      as author_password,
    A.email         as author_email,
    A.bio           as author_bio,
    CA.id           as co_author_id,
    CA.username     as co_author_username,
    CA.password     as co_author_password,
    CA.email        as co_author_email,
    CA.bio          as co_author_bio
  from Blog B
  left outer join Author A on B.author_id = A.id
  left outer join Author CA on B.co_author_id = CA.id
  where B.id = #{id}
</select>
```

Recall that the resultMap for Author is defined as follows.

```
<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>
```

Because the column names in the results differ from the columns defined in the resultMap, you need to specify `columnPrefix` to reuse the resultMap for mapping co-author results.

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author"
    resultMap="authorResult" />
  <association property="coAuthor"
    resultMap="authorResult"
    columnPrefix="co_" />
</resultMap>
```

上面你已经看到了如何处理“有一个”类型关联。但是“有很多个”是怎样的?下面这 个部分就是来讨论这个主题的。

集合

```
<collection property="posts" ofType="domain.blog.Post">
  <id property="id" column="post_id"/>
  <result property="subject" column="post_subject"/>
  <result property="body" column="post_body"/>
</collection>
```

集合元素的作用几乎和关联是相同的。实际上,它们也很相似,文档的异同是多余的。 所以我们更多关注于它们的不同。

我们来继续上面的示例,一个博客只有一个作者。但是博客有很多文章。在博客类中, 这可以由下面这样的写法来表示:

```
private List<Post> posts;
```

要映射嵌套结果集合到 List 中,我们使用集合元素。就像关联元素一样,我们可以从连接中使用嵌套查询,或者嵌套结果。

集合的嵌套查询

首先,让我们看看使用嵌套查询来为博客加载文章。

```
<resultMap id="blogResult" type="Blog">
  <collection property="posts" javaType="ArrayList" column="id"
ofType="Post" select="selectPostsForBlog"/>
</resultMap>

<select id="selectBlog" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectPostsForBlog" resultType="Blog">
  SELECT * FROM POST WHERE BLOG_ID = #{id}
</select>
```

这里你应该注意很多东西,但大部分代码和上面的关联元素是非常相似的。首先,你应 该注意我们使用的是集合元素。然后要注意那个新的“ofType”属性。这个属性用来区分 JavaBean(或字段)属性类型和集合包含的类型来说是很重要的。所以你可以读出下面这个 映射:

```
<collection property="posts" javaType="ArrayList" column="id" of
Type="Post" select="selectPostsForBlog"/>
```

读作: “在 Post 类型的 ArrayList 中的 posts 的集合。”

javaType 属性是不需要的,因为 MyBatis 在很多情况下会为你算出来。所以你可以缩短 写法:

```
<collection property="posts" column="id" ofType="Post" select="s
electPostsForBlog"/>
```

集合的嵌套结果

至此,你可以猜测集合的嵌套结果是如何来工作的,因为它和关联完全相同,除了它应用了一个“ofType”属性

First, let's look at the SQL:

```
<select id="selectBlog" resultMap="blogResult">
  select
  B.id as blog_id,
  B.title as blog_title,
  B.author_id as blog_author_id,
  P.id as post_id,
  P.subject as post_subject,
  P.body as post_body,
  from Blog B
  left outer join Post P on B.id = P.blog_id
  where B.id = #{id}
</select>
```

我们又一次联合了博客表和文章表,而且关注于保证特性,结果列标签的简单映射。现 在用文章映射集合映射博客,可以简单写为:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
  </collection>
</resultMap>
```

同样,要记得 id 元素的重要性,如果你不记得了,请阅读上面的关联部分。

同样, 如果你引用更长的形式允许你的结果映射的更多重用, 你可以使用下面这个替代 的映射:

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post" resultMap="blogPost
Result" columnPrefix="post_"/>
</resultMap>

<resultMap id="blogPostResult" type="Post">
  <id property="id" column="id"/>
  <result property="subject" column="subject"/>
  <result property="body" column="body"/>
</resultMap>
```

注意 这个对你所映射的内容没有深度,广度或关联和集合相联合的限制。当映射它们 时你应该在大脑中保留它们的表现。 你的应用在找到最佳方法前要一直进行的单元测试和性 能测试。好在 myBatis 让你后来可以改变想法,而不对你的代码造成

很小(或任何)影响。

高级关联和集合映射是一个深度的主题。文档只能给你介绍到这了。加上一点联系,你 会很快清楚它们的用法。

鉴别器

```
<discriminator javaType="int" column="draft">
  <case value="1" resultType="DraftPost"/>
</discriminator>
```

有时一个单独的数据库查询也许返回很多不同 (但是希望有些关联) 数据类型的结果集。 鉴别器元素就是被设计来处理这个情况的, 还有包括类的继承层次结构。 鉴别器非常容易理 解,因为它的表现很像 Java 语言中的 switch 语句。

定义鉴别器指定了 column 和 javaType 属性。 列是 MyBatis 查找比较值的地方。 JavaType 是需要被用来保证等价测试的合适类型(尽管字符串在很多情形下都会有用)。比如:

```
<resultMap id="vehicleResult" type="Vehicle">
  <id property="id" column="id" />
  <result property="vin" column="vin"/>
  <result property="year" column="year"/>
  <result property="make" column="make"/>
  <result property="model" column="model"/>
  <result property="color" column="color"/>
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultMap="carResult"/>
    <case value="2" resultMap="truckResult"/>
    <case value="3" resultMap="vanResult"/>
    <case value="4" resultMap="suvResult"/>
  </discriminator>
</resultMap>
```

在这个示例中, MyBatis 会从结果集中得到每条记录, 然后比较它的 vehicle 类型的值。 如果它匹配任何一个鉴别器的实例,那么就使用这个实例指定的结果映射。换句话说,这样 做完全是剩余的结果映射被忽略(除非它被扩展,这在第二个示例中讨论) 。如果没有任何 一个实例相匹配,那么 MyBatis 仅仅使用鉴别器块外定义的结果映射。所以,如果 carResult 按如下声明:

```
<resultMap id="carResult" type="Car">
  <result property="doorCount" column="door_count" />
</resultMap>
```

那么只有 doorCount 属性会被加载。这步完成后完整地允许鉴别器实例的独立组,尽管 和父结果映射可能没有什么关系。这种情况下,我们当然知道 cars 和 vehicles 之间有关系, 如 Car 是一个 Vehicle 实例。因此,我们想要剩余的属性也被加载。我们设置的结果映射的 简单改变如下。

```
<resultMap id="carResult" type="Car" extends="vehicleResult">
  <result property="doorCount" column="door_count" />
</resultMap>
```

现在 vehicleResult 和 carResult 的属性都会被加载了。

尽管曾经有些人会发现这个外部映射定义会多少有一些令人厌烦之处。 因此还有另外一 种语法来做简洁的映射风格。比如:

```
<resultMap id="vehicleResult" type="Vehicle">
  <id property="id" column="id" />
  <result property="vin" column="vin"/>
  <result property="year" column="year"/>
  <result property="make" column="make"/>
  <result property="model" column="model"/>
  <result property="color" column="color"/>
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultType="carResult">
      <result property="doorCount" column="door_count" />
    </case>
    <case value="2" resultType="truckResult">
      <result property="boxSize" column="box_size" />
      <result property="extendedCab" column="extended_cab" />
    </case>
    <case value="3" resultType="vanResult">
      <result property="powerSlidingDoor" column="power_sliding_
door" />
    </case>
    <case value="4" resultType="suvResult">
      <result property="allWheelDrive" column="all_wheel_drive"
/>
    </case>
  </discriminator>
</resultMap>
```

要记得 这些都是结果映射, 如果你不指定任何结果, 那么 MyBatis 将会为你自动匹配列 和属性。所以这些例子中的大部分是很冗长的,而其实是不需要的。也就是说,很多数据库 是很复杂的,我们不太可能对所有示例都能依靠它。

自动映射

正如你在前面一节看到的，在简单的场景下，MyBatis可以替你自动映射查询结果。 如果遇到复杂的场景，你需要构建一个result map。 但是在本节你将看到，你也可以混合使用这两种策略。 让我们到深一点的层面上看看自动映射是怎样工作的。

当自动映射查询结果时，MyBatis会获取sql返回的列名并在java类中查找相同名字的属性（忽略大小写）。 这意味着如果Mybatis发现了*ID*列和*id*属性，Mybatis会将*ID*的值赋给*id*。

通常数据库列使用大写单词命名，单词间用下划线分隔；而java属性一般遵循驼峰命名法。 为了在这两种命名方式之间启用自动映射，需要将 `mapUnderscoreToCamelCase` 设置为true。

自动映射甚至在特定的result map下也能工作。在这种情况下，对于每一个result map,所有的ResultSet提供的列， 如果没有被手工映射，则将被自动映射。自动映射处理完毕后手工映射才会被处理。 在接下来的例子中， *id* 和 *userName*列将被自动映射， *hashed_password* 列将根据配置映射。

```
<select id="selectUsers" resultMap="userResultMap">
  select
    user_id             as "id",
    user_name           as "userName",
    hashed_password
  from some_table
  where id = #{id}
</select>
```

```
<resultMap id="userResultMap" type="User">
  <result property="password" column="hashed_password"/>
</resultMap>
```

There are three auto-mapping levels:

- `NONE` - disables auto-mapping. Only manually mapped properties will be set.
- `PARTIAL` - will auto-map results except those that have nested result mappings defined inside (joins).
- `FULL` - auto-maps everything.

The default value is `PARTIAL`, and it is so for a reason. When `FULL` is used auto-mapping will be performed when processing join results and joins retrieve data of several different entities in the same row hence this may result in undesired mappings. To understand the risk have a look at the following sample:

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id,
    B.title,
    A.username,
  from Blog B left outer join Author A on B.author_id = A.id
  where B.id = #{id}
</select>
```

```
<resultMap id="blogResult" type="Blog">
  <association property="author" resultMap="authorResult"/>
</resultMap>

<resultMap id="authorResult" type="Author">
  <result property="username" column="author_username"/>
</resultMap>
```

With this result map both *Blog* and *Author* will be auto-mapped. But note that *Author* has an *id* property and there is a column named *id* in the ResultSet so Author's id will be filled with Blog's id, and that is not what you were expecting. So use the `FULL` option with caution.

Regardless of the auto-mapping level configured you can enable or disable the automapping for an specific ResultMap by adding the attribute `autoMapping` to it:

```
<resultMap id="userResultMap" type="User" autoMapping="false">
  <result property="password" column="hashed_password"/>
</resultMap>
```

缓存

MyBatis 包含一个非常强大的查询缓存特性,它可以非常方便地配置和定制。MyBatis 3 中的缓存实现的很多改进都已经实现了,使得它更加强大而且易于配置。

默认情况下是没有开启缓存的,除了局部的 session 缓存,可以增强变现而且处理循环 依赖也是必须的。要开启二级缓存,你需要在你的 SQL 映射文件中添加一行:

```
<cache/>
```

字面上看就是这样。这个简单语句的效果如下:

- 映射语句文件中的所有 select 语句将会被缓存。
- 映射语句文件中的所有 insert,update 和 delete 语句会刷新缓存。
- 缓存会使用 Least Recently Used(LRU,最近最少使用的)算法来收回。

- 根据时间表(比如 no Flush Interval,没有刷新间隔), 缓存不会以任何时间顺序来刷新。
- 缓存会存储列表集合或对象(无论查询方法返回什么)的 1024 个引用。
- 缓存会被视为是 read/write(可读/可写)的缓存,意味着对象检索不是共享的,而且可以安全地被调用者修改,而不干扰其他调用者或线程所做的潜在修改。

所有的这些属性都可以通过缓存元素的属性来修改。比如:

```
<cache
  eviction="FIFO"
  flushInterval="60000"
  size="512"
  readOnly="true"/>
```

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会 导致冲突。

可用的收回策略有:

- `LRU` – 最近最少使用的:移除最长时间不被使用的对象。
- `FIFO` – 先进先出:按对象进入缓存的顺序来移除它们。
- `SOFT` – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
- `WEAK` – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

默认的是 LRU。

flushInterval(刷新间隔)可以被设置为任意的正整数,而且它们代表一个合理的毫秒形式的时间段。默认情况是不设置,也就是没有刷新间隔,缓存仅仅调用语句时刷新。

size(引用数目)可以被设置为任意正整数,要记住你缓存的对象数目和你运行环境的可用内存资源数目。默认值是 1024。

readOnly(只读)属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。可读写的缓存 会返回缓存对象的拷贝(通过序列化) 。这会慢一些,但是安全,因此默认是 false。

使用自定义缓存

除了这些自定义缓存的方式, 你也可以通过实现你自己的缓存或为其他第三方缓存方案 创建适配器来完全覆盖缓存行为。

```
<cache type="com.domain.something.MyCustomCache"/>
```

这个示 例展 示了 如何 使用 一个 自定义 的缓 存实 现。type 属 性指 定的 类必 须实现 org.mybatis.cache.Cache 接口。这个接口是 MyBatis 框架中很多复杂的接口之一,但是简单 给定它做什么就行。

```
public interface Cache {
  String getId();
  int getSize();
  void putObject(Object key, Object value);
  Object getObject(Object key);
  boolean hasKey(Object key);
  Object removeObject(Object key);
  void clear();
}
```

要配置你的缓存, 简单和公有的 JavaBeans 属性来配置你的缓存实现, 而且是通过 cache 元素来传递属性, 比如, 下面代码会在你的缓存实现中调用一个称为 “setCacheFile(String file)” 的方法:

```
<cache type="com.domain.something.MyCustomCache">
  <property name="cacheFile" value="/tmp/my-custom-cache.tmp"/>
</cache>
```

你可以使用所有简单类型作为 JavaBeans 的属性,MyBatis 会进行转换。

记得缓存配置和缓存实例是绑定在 SQL 映射文件的命名空间是很重要的。因此,所有 在相同命名空间的语句正如绑定的缓存一样。 语句可以修改和缓存交互的方式,或在语句的 语句的基础上使用两种简单的属性来完全排除它们。默认情况下,语句可以这样来配置:

```
<select ... flushCache="false" useCache="true"/>
<insert ... flushCache="true"/>
<update ... flushCache="true"/>
<delete ... flushCache="true"/>
```

因为那些是默认的,你明显不能明确地以这种方式来配置一条语句。相反,如果你想改 变默认的行为,只能设置 flushCache 和 useCache 属性。比如,在一些情况下你也许想排除 从缓存中查询特定语句结果,或者你也许想要一个查询语句来刷新缓存。相似地,你也许有 一些更新语句依靠执行而不需要刷新缓存。

参照缓存

回想一下上一节内容, 这个特殊命名空间的唯一缓存会被使用或者刷新相同命名空间内 的语句。也许将来的某个时候,你会想在命名空间中共享相同的缓存配置和实例。在这样的 情况下你可以使用 cache-ref 元素来引用另外一个缓存。

```
<cache-ref namespace="com.someone.application.data.SomeMapper"/>
```

动态 SQL

MyBatis 的强大特性之一便是它的动态 SQL。如果你有使用 JDBC 或其他类似框架的经验，你就能体会到根据不同条件拼接 SQL 语句有多么痛苦。拼接的时候要确保不能忘了必要的空格，还要注意省掉列名列表最后的逗号。利用动态 SQL 这一特性可以彻底摆脱这种痛苦。

通常使用动态 SQL 不可能是独立的一部分,MyBatis 当然使用一种强大的动态 SQL 语言来改进这种情形,这种语言可以被用在任意的 SQL 映射语句中。

动态 SQL 元素和使用 JSTL 或其他类似基于 XML 的文本处理器相似。在 MyBatis 之前的版本中,有很多的元素需要来了解。MyBatis 3 大大提升了它们,现在用不到原先一半的元素就可以了。MyBatis 采用功能强大的基于 OGNL 的表达式来消除其他元素。

- if
- choose (when, otherwise)
- trim (where, set)
- foreach

if

动态 SQL 通常要做的事情是有条件地包含 where 子句的一部分。比如:

```
<select id="findActiveBlogWithTitleLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
</select>
```

这条语句提供了一个可选的文本查找类型的功能。如果没有传入“title”，那么所有处于“ACTIVE”状态的BLOG都会返回；反之若传入了“title”，那么就会把模糊查找“title”内容的BLOG结果返回（就这个例子而言，细心的读者会发现其中的参数值是可以包含一些掩码或通配符的）。

如果想可选地通过“title”和“author”两个条件搜索该怎么办呢？首先，改变语句的名称让它更具实际意义；然后只要加入另一个条件即可。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

choose, when, otherwise

有些时候，我们不想用到所有的条件语句，而只想从中择其一二。针对这种情况，MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

还是上面的例子，但是这次变为提供了“title”就按“title”查找，提供了“author”就按“author”查找，若两者都没有提供，就返回所有符合条件的BLOG（实际情况可能是由管理员按一定策略选出BLOG列表，而不是返回大量无意义的随机结果）。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

trim, where, set

前面几个例子已经合宜地解决了一个臭名昭著的动态 SQL 问题。现在考虑回到“if”示例，这次我们将“ACTIVE = 1”也设置成动态的条件，看看会发生什么。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE
  <if test="state != null">
    state = #{state}
  </if>
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

如果这些条件没有一个能匹配上将会怎样？最终这条 SQL 会变成这样：

```
SELECT * FROM BLOG
WHERE
```

这会导致查询失败。如果仅仅第二个条件匹配又会怎样？这条 SQL 最终会是这样:

```
SELECT * FROM BLOG
WHERE
AND title like ‘someTitle’
```

这个查询也会失败。这个问题不能简单的用条件句式来解决，如果你也曾经被迫这样写过，那么你很可能从此以后都不想再这样去写了。

MyBatis 有一个简单的处理，这在90%的情况下都会有用。而在不能使用的地方，你可以自定义处理方式来令其正常工作。一处简单的修改就能得到想要的效果：

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

where 元素知道只有在一个以上的if条件有值的情况下才去插入“WHERE”子句。而且，若最后的内容是“AND”或“OR”开头的，where 元素也知道如何将他们去除。

如果 where 元素没有按正常套路出牌，我们还是可以通过自定义 trim 元素来定制我们想要的功能。比如，和 where 元素等价的自定义 trim 元素为：

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>
```

prefixOverrides 属性会忽略通过管道分隔的文本序列（注意此例中的空格也是必要的）。它带来的结果就是所有在 prefixOverrides 属性中指定的内容将被移除，并且插入 prefix 属性中指定的内容。

类似的用于动态更新语句的解决方案叫做 set。set 元素可以被用于动态包含需要更新的列，而舍去其他的。比如：

```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

这里，set 元素会动态前置 SET 关键字，同时也会消除无关的逗号，因为用了条件语句之后很可能就会在生成的赋值语句的后面留下这些逗号。

若你对等价的自定义 trim 元素的样子感兴趣，那这就应该是它的真面目：

```
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```

注意这里我们忽略的是后缀中的值，而又一次附加了前缀中的值。

foreach

动态 SQL 的另外一个常用的必要操作是需要对一个集合进行遍历，通常是在构建 IN 条件语句的时候。比如：

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
  <foreach item="item" index="index" collection="list"
      open="(" separator="," close=")">
        #{item}
  </foreach>
</select>
```

foreach 元素的功能是非常强大的，它允许你指定一个集合，声明可以用在元素体内的集合项和索引变量。它也允许你指定开闭匹配的字符串以及在迭代中间放置分隔符。这个元素是很智能的，因此它不会偶然地附加多余的分隔符。

注意 你可以将一个 List 实例或者数组作为参数对象传给 MyBatis，当你这么做的时候，MyBatis 会自动将它包装在一个 Map 中并以名称为键。List 实例将会以“list”作为键，而数组实例的键将是“array”。

到此我们已经完成了涉及 XML 配置文件和 XML 映射文件的讨论。下一部分将详细探讨 Java API，这样才能从已创建的映射中获取最大利益。

bind

`bind` 元素可以从 OGNL 表达式中创建一个变量并将其绑定到上下文。比如：

```
<select id="selectBlogsLike" resultType="Blog">
  <bind name="pattern" value="'%' + _parameter.getTitle() + '%'"
 />
  SELECT * FROM BLOG
  WHERE title LIKE #{pattern}
</select>
```

Multi-db vendor support

一个配置了“_databaseId”变量的 databaseIdProvider 对于动态代码来说是可用的，这样就可以根据不同的数据库厂商构建特定的语句。比如下面的例子：

```
<insert id="insert">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    <if test="_databaseId == 'oracle'">
      select seq_users.nextval from dual
    </if>
    <if test="_databaseId == 'db2'">
      select nextval for seq_users from sysibm.sysdummy1"
    </if>
  </selectKey>
  insert into users values (#{id}, #{name})
</insert>
```

动态 SQL 中可插拔的脚本语言

MyBatis 从 3.2 开始支持可插拔的脚本语言，因此你可以在插入一种语言的驱动（language driver）之后来写基于这种语言的动态 SQL 查询。

可以通过实现下面接口的方式来插入一种语言：

```
public interface LanguageDriver {
  ParameterHandler createParameterHandler(MappedStatement mapped
Statement, Object parameterObject, BoundSql boundSql);
  SqlSource createSqlSource(Configuration configuration, XNode s
cript, Class<?> parameterType);
  SqlSource createSqlSource(Configuration configuration, String
script, Class<?> parameterType);
}
```

一旦有了自定义的语言驱动，你就可以在 mybatis-config.xml 文件中将它设置为默认语言：

```
<typeAliases>
  <typeAlias type="org.sample.MyLanguageDriver" alias="myLanguag
e"/>
</typeAliases>
<settings>
  <setting name="defaultScriptingLanguage" value="myLanguage"/>
</settings>
```

除了设置默认语言，你也可以针对特殊的语句指定特定语言，这可以通过如下的 `lang` 属性来完成：

```
<select id="selectBlog" lang="myLanguage">
  SELECT * FROM BLOG
</select>
```

或者在你正在使用的映射中加上注解 `@Lang` 来完成：

```
public interface Mapper {
  @Lang(MyLanguageDriver.class)
  @Select("SELECT * FROM BLOG")
  List<Blog> selectBlog();
}
```

注意 可以将 Apache Velocity 作为动态语言来使用，更多细节请参考 MyBatis-Velocity 项目。

你前面看到的所有 xml 标签都是默认 MyBatis 语言提供的，它是由别名为 `xml` 语言驱动器 `org.apache.ibatis.scripting.xmltags.XmlLanguageDriver` 驱动的。

Java API

既然你已经知道如何配置 MyBatis 和创建映射文件,你就已经准备好来提升技能了。MyBatis 的 Java API 就是你收获你所做的努力的地方。正如你即将看到的,和 JDBC 相比, MyBatis 很大程度简化了你的代码而且保持简洁,很容易理解和维护。MyBatis 3 已经引入 了很多重要的改进来使得 SQL 映射更加优秀。

应用目录结构

在我们深入 Java API 之前,理解关于目录结构的最佳实践是很重要的。MyBatis 非常灵 活, 你可以用你自己的文件来做几乎所有的事情。 但是对于任一框架, 都有一些最佳的方式。

让我们看一下典型应用的目录结构:

```
/my_application
  /bin
  /devlib
  **/lib                `<-- MyBatis *.jar文件在这里。`**
  /src
    /org/myapp/
      /action
      **/data           `<-- MyBatis配置文件在这里, 包括映射器类, XML配置, XML映射文件。`**
        /mybatis-config.xml
        /BlogMapper.java
        /BlogMapper.xml
      /model
      /service
      /view
    **/properties       `<-- 在你XML中配置的属性 文件在这里。`**
  /test
    /org/myapp/
      /action
      /data
      /model
      /service
      /view
    /properties
  /web
    /WEB-INF
      /web.xml
```

Remember, these are preferences, not requirements, but others will thank you for using a common directory structure.

这部分内容剩余的示例将假设你使用了这种目录结构。

SqlSessions

使用 MyBatis 的主要 Java 接口就是 SqlSession。尽管你可以使用这个接口执行命令,获 取映射器和管理事务。我们会讨论 SqlSession 本身更多,但是首先我们还是要了解如果获取 一个 SqlSession 实例。SqlSessions 是由 SqlSessionFactory 实例创建的。SqlSessionFactory 对 象 包 含 创 建 SqlSession 实 例 的 所 有 方 法 。 而 SqlSessionFactory 本 身 是 由 SqlSessionFactoryBuilder 创建的,它可以从 XML 配置,注解或手动配置 Java 来创建 SqlSessionFactory。

NOTE When using MyBatis with a dependency injection framework like Spring or Guice, SqlSessions are created and injected by the DI framework so you don't need to use the SqlSessionFactoryBuilder or SqlSessionFactory and can go directly to the SqlSession section. Please refer to the MyBatis-Spring or MyBatis-Guice manuals for further info.

SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 有五个 build()方法,每一种都允许你从不同的资源中创建一个 SqlSession 实例。

```
SqlSessionFactory build(InputStream inputStream)
SqlSessionFactory build(InputStream inputStream, String environm
ent)
SqlSessionFactory build(InputStream inputStream, Properties prop
erties)
SqlSessionFactory build(InputStream inputStream, String env, Pro
perties props)
SqlSessionFactory build(Configuration config)
```

第一种方法是最常用的,它使用了一个参照了 XML 文档或上面讨论过的更特定的 mybatis-config.xml 文件的 Reader 实例。 可选的参数是 environment 和 properties。 Environment 决定加载哪种环境,包括数据源和事务管理器。比如:

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
        ...
    <dataSource type="POOLED">
        ...
  </environment>
  <environment id="production">
    <transactionManager type="MANAGED">
        ...
    <dataSource type="JNDI">
        ...
  </environment>
</environments>
```

如果你调用了 一个使用 environment 参数 方 式的 build 方法, 那么 MyBatis 将会使用 configuration 对象来配置这个 environment。 当然, 如果你指定了一个不合法的 environment, 你会得到错误提示。 如果你调用了其中之一没有 environment 参数的 build 方法, 那么就使用 默认的 environment(在上面的示例中就会指定为 default=”development”)。

如果你调用了使用 properties 实例的方法,那么 MyBatis 就会加载那些 properties(属性 配置文件) ,并你在你配置中可使用它们。那些属性可以用 ${propName}语法形式多次用在 配置文件中。

回想一下,属性可以从 mybatis-config.xml 中被引用,或者直接指定它。因此理解优先级是很重要的。我们在文档前面已经提及它了,但是这里要再次重申:

如果一个属性存在于这些位置,那么 MyBatis 将会按找下面的顺序来加载它们:

- 在 properties 元素体中指定的属性首先被读取,
- 从 properties 元素的类路径 resource 或 url 指定的属性第二个被读取, 可以覆盖已经 指定的重复属性,
- 作为方法参 数传递 的属性最 后被读 取,可以 覆盖已 经从 properties 元 素体和 resource/url 属性中加载的任意重复属性。

因此,最高优先级的属性是通过方法参数传递的,之后是 resource/url 属性指定的,最后是在 properties 元素体中指定的属性。

总结一下,前四个方法很大程度上是相同的,但是由于可以覆盖,就允许你可选地指定 environment 和/或 properties。 这里给出一个从 mybatis-config.xml 文件创建 SqlSessionFactory 的示例:

```
String **resource** = "org/mybatis/builder/mybatis-config.xml";
InputStream **inputStream** = Resources.getResourceAsStream(reso
urce);
SqlSessionFactoryBuilder **builder** = new SqlSessionFactoryBuil
der();
SqlSessionFactory **factory** = builder.build(inputStream);
```

注意这里我们使用了 Resources 工具类,这个类在 org.mybatis.io 包中。Resources 类正 如其名,会帮助你从类路径下,文件系统或一个 web URL 加载资源文件。看一下这个类的 源代码或者通过你的 IDE 来查看,就会看到一整套有用的方法。这里给出一个简表:

```
URL getResourceURL(String resource)
URL getResourceURL(ClassLoader loader, String resource)
InputStream getResourceAsStream(String resource)
InputStream getResourceAsStream(ClassLoader loader, String resou
rce)
Properties getResourceAsProperties(String resource)
Properties getResourceAsProperties(ClassLoader loader, String re
source)
Reader getResourceAsReader(String resource)
Reader getResourceAsReader(ClassLoader loader, String resource)
File getResourceAsFile(String resource)
File getResourceAsFile(ClassLoader loader, String resource)
InputStream getUrlAsStream(String urlString)
Reader getUrlAsReader(String urlString)
Properties getUrlAsProperties(String urlString)
Class classForName(String className)
```

最后一个 build 方法使用了一个 Configuration 实例。configuration 类包含你可能需要了 解 SqlSessionFactory 实例的所有内容。Configuration 类对于配置的自查很有用,包含查找和 操作 SQL 映射(不推荐使用,因为应用正接收请求) 。configuration 类有所有配置的开关, 这些你已经了解了,只在 Java API 中露出来。这里有一个简单的示例,如何手动配置 configuration 实例,然后将它传递给 build()方法来创建 SqlSessionFactory。

```
DataSource dataSource = BaseDataTest.createBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFacto
ry();

Environment environment = new Environment("development", transac
tionFactory, dataSource);

Configuration configuration = new Configuration(environment);
configuration.setLazyLoadingEnabled(true);
configuration.setEnhancementEnabled(true);
configuration.getTypeAliasRegistry().registerAlias(Blog.class);
configuration.getTypeAliasRegistry().registerAlias(Post.class);
configuration.getTypeAliasRegistry().registerAlias(Author.class)
;
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);

SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder(
);
SqlSessionFactory factory = builder.build(configuration);
```

现在你有一个 SqlSessionFactory,可以用来创建 SqlSession 实例。

SqlSessionFactory

SqlSessionFactory 有六个方法可以用来创建 SqlSession 实例。通常来说,如何决定是你 选择下面这些方法时:

- **Transaction (事务)**: 你想为 session 使用事务或者使用自动提交(通常意味着很多 数据库和/或 JDBC 驱动没有事务)?
- **Connection (连接)**: 你想 MyBatis 获得来自配置的数据源的连接还是提供你自己
- **Execution (执行)**: 你想 MyBatis 复用预处理语句和/或批量更新语句(包括插入和 删除)?

重载的 openSession()方法签名设置允许你选择这些可选中的任何一个组合。

```
SqlSession openSession()
SqlSession openSession(boolean autoCommit)
SqlSession openSession(Connection connection)
SqlSession openSession(TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType,TransactionIsolatio
nLevel level)
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCommit
)
SqlSession openSession(ExecutorType execType, Connection connect
ion)
Configuration getConfiguration();
```

默认的 openSession()方法没有参数,它会创建有如下特性的 SqlSession:

- 会开启一个事务(也就是不自动提交)
- 连接对象会从由活动环境配置的数据源实例中得到。
- 事务隔离级别将会使用驱动或数据源的默认设置。
- 预处理语句不会被复用,也不会批量处理更新。

这些方法大都可以自我解释的。 开启自动提交, “true” 传递 给可选的 autoCommit 参数。 提供自定义的连接,传递一个 Connection 实例给 connection 参数。注意没有覆盖同时设置 Connection 和 autoCommit 两者的方法,因为 MyBatis 会使用当前 connection 对象提供的设 置。 MyBatis 为事务隔离级别调用使用一个 Java 枚举包装器, 称为 TransactionIsolationLevel, 否则它们按预期的方式来工作,并有 JDBC 支持的 5 级 (NONE,READ_UNCOMMITTED,READ_COMMITTED,REPEA TABLE_READ,SERIALIZA BLE)

还有一个可能对你来说是新见到的参数,就是 ExecutorType。这个枚举类型定义了 3 个 值:

- `ExecutorType.SIMPLE`: 这个执行器类型不做特殊的事情。它为每个语句的执行创建一个新的预处理语句。
- `ExecutorType.REUSE`: 这个执行器类型会复用预处理语句。
- `ExecutorType.BATCH`: 这个执行器会批量执行所有更新语句,如果 SELECT 在它们中间执行还会标定它们是 必须的,来保证一个简单并易于理解的行为。

注意 在 SqlSessionFactory 中还有一个方法我们没有提及,就是 getConfiguration()。这 个方法会返回一个 Configuration 实例,在运行时你可以使用它来自检 MyBatis 的配置。

注意 如果你已经使用之前版本 MyBatis,你要回忆那些 session,transaction 和 batch 都是分离的。现在和以往不同了,这些都包含在 session 的范围内了。你需要处理分开处理 事务或批量操作来得到它们的效果。

SqlSession

如上面所提到的,SqlSession 实例在 MyBatis 中是非常强大的一个类。在这里你会发现 所有执行语句的方法,提交或回滚事务,还有获取映射器实例。

在 SqlSession 类中有超过 20 个方法,所以将它们分开成易于理解的组合。

语句执行方法

这些方法被用来执行定义在 SQL 映射的 XML 文件中的 SELECT,INSERT,UPDA E T 和 DELETE 语句。它们都会自行解释,每一句都使用语句的 ID 属性和参数对象,参数可以 是原生类型(自动装箱或包装类) ,JavaBean,POJO 或 Map。

```
<T> T selectOne(String statement, Object parameter)
<E> List<E> selectList(String statement, Object parameter)
<K,V> Map<K,V> selectMap(String statement, Object parameter, Str
ing mapKey)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

selectOne 和 selectList 的不同仅仅是 selectOne 必须返回一个对象。 如果多余一个, 或者 没有返回 (或返回了 null) 那么就会抛出异常。 , 如果你不知道需要多少对象, 使用 selectList。

如果你想检查一个对象是否存在,那么最好返回统计数(0 或 1) 。因为并不是所有语句都需 要参数,这些方法都是有不同重载版本的,它们可以不需要参数对象。

```
<T> T selectOne(String statement)
<E> List<E> selectList(String statement)
<K,V> Map<K,V> selectMap(String statement, String mapKey)
int insert(String statement)
int update(String statement)
int delete(String statement)
```

最后,还有查询方法的三个高级版本,它们允许你限制返回行数的范围,或者提供自定义结果控制逻辑,这通常用于大量的数据集合。

```
<E> List<E> selectList (String statement, Object parameter, RowB
ounds rowBounds)
<K,V> Map<K,V> selectMap(String statement, Object parameter, Str
ing mapKey, RowBounds rowbounds)
void select (String statement, Object parameter, ResultHandler<T
> handler)
void select (String statement, Object parameter, RowBounds rowBo
unds, ResultHandler<T> handler)
```

RowBounds 参数会告诉 MyBatis 略过指定数量的记录,还有限制返回结果的数量。RowBounds 类有一个构造方法来接收 offset 和 limit,否则是不可改变的。

```
int offset = 100;
int limit = 25;
RowBounds rowBounds = new RowBounds(offset, limit);
```

不同的驱动会实现这方面的不同级别的效率。对于最佳的表现,使用结果集类型的 SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE(或句话说:不是 FORWARD_ONLY)。

ResultHandler 参数允许你按你喜欢的方式处理每一行。你可以将它添加到 List 中,创 建 Map, 或抛出每个结果而不是只保留总计。 Set 你可以使用 ResultHandler 做很多漂亮的事, 那就是 MyBatis 内部创建结果集列表。

它的接口很简单。

```
package org.apache.ibatis.session;
public interface ResultHandler<T> {
  void handleResult(ResultContext<? extends T> context);
}
```

ResultContext 参数给你访问结果对象本身的方法, 大量结果对象被创建, 你可以使用布 尔返回值的 stop()方法来停止 MyBatis 加载更多的结果。

Batch update statement Flush Method

There is method for flushing(executing) batch update statements that stored in a JDBC driver class at any timing. This method can be used when you use the `ExecutorType.BATCH` as `ExecutorType` .

```
List<BatchResult> flushStatements()
```

事务控制方法

控制事务范围有四个方法。 当然, 如果你已经选择了自动提交或你正在使用外部事务管 理器,这就没有任何效果了。然而,如果你正在使用 JDBC 事务管理员,由 Connection 实 例来控制,那么这四个方法就会派上用场:

```
void commit()
void commit(boolean force)
void rollback()
void rollback(boolean force)
```

默认情况下 MyBatis 不会自动提交事务, 除非它侦测到有插入, 更新或删除操作改变了 数据库。如果你已经做出了一些改变而没有使用这些方法,那么你可以传递 true 到 commit 和 rollback 方法来保证它会被提交(注意,你不能在自动提交模式下强制 session,或者使用 了外部事务管理器时) 。很多时候你不用调用 rollback(),因为如果你没有调用 commit 时 MyBatis 会替你完成。然而,如果你需要更多对多提交和回滚都可能的 session 的细粒度控 制,你可以使用回滚选择来使它成为可能。

NOTE MyBatis-Spring and MyBatis-Guice provide declarative transaction handling. So if you are using MyBatis with Spring or Guice please refer to their specific manuals.

清理 **Session** 级的缓存

```
void clearCache()
```

SqlSession 实例有一个本地缓存在执行 update,commit,rollback 和 close 时被清理。要 明确地关闭它(获取打算做更多的工作) ,你可以调用 clearCache()。

确保 **SqlSession** 被关闭

```
void close()
```

你必须保证的最重要的事情是你要关闭所打开的任何 session。保证做到这点的最佳方 式是下面的工作模式:

```
SqlSession session = sqlSessionFactory.openSession();
try {
    // following 3 lines pseudocod for "doing some work"
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
} finally {
    session.close();
}
```

还有，如果你正在使用jdk 1.7以上的版本还有MyBatis 3.2以上的版本，你可以使用try-with-resources语句：

```
try (SqlSession session = sqlSessionFactory.openSession()) {
    // following 3 lines pseudocode for "doing some work"
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
}
```

注意 就像 SqlSessionFactory,你可以通过调用 getConfiguration()方法获得 SqlSession 使用的 Configuration 实例

```
Configuration getConfiguration()
```

使用映射器

```
<T> T getMapper(Class<T> type)
```

上述的各个 insert,update,delete 和 select 方法都很强大,但也有些繁琐,没有类型安全,对于你的 IDE 也没有帮助,还有可能的单元测试。在上面的入门章节中我们已经看到 了一个使用映射器的示例。

因此, 一个更通用的方式来执行映射语句是使用映射器类。 一个映射器类就是一个简单 的接口,其中的方法定义匹配于 SqlSession 方法。下面的示例展示了一些方法签名和它们是 如何映射到 SqlSession 的。

```
public interface AuthorMapper {
  // (Author) selectOne("selectAuthor",5);
  Author selectAuthor(int id);
  // (List<Author>) selectList(“selectAuthors”)
  List<Author> selectAuthors();
  // (Map<Integer,Author>) selectMap("selectAuthors", "id")
  @MapKey("id")
  Map<Integer, Author> selectAuthors();
  // insert("insertAuthor", author)
  int insertAuthor(Author author);
  // updateAuthor("updateAuthor", author)
  int updateAuthor(Author author);
  // delete("deleteAuthor",5)
  int deleteAuthor(int id);
}
```

总之, 每个映射器方法签名应该匹配相关联的 SqlSession 方法, 而没有字符串参数 ID。 相反,方法名必须匹配映射语句的 ID。

此外,返回类型必须匹配期望的结果类型。所有常用的类型都是支持的,包括:原生类型,Map,POJO 和 JavaBean。

映射器接口不需要去实现任何接口或扩展任何类。 只要方法前面可以被用来唯一标识对 应的映射语句就可以了。

映射器接口可以扩展其他接口。当使用 XML 来构建映射器接口时要保证在合适的命名 空间中有语句。 而且, 唯一的限制就是你不能在两个继承关系的接口中有相同的方法签名 (这 也是不好的想法)。

你可以传递多个参数给一个映射器方法。 如果你这样做了, 默认情况下它们将会以它们 在参数列表中的位置来命名,比如:#{param1},#{param2}等。如果你想改变参数的名称(只在多参数 情况下) ,那么你可以在参数上使用@Param(“paramName”)注解。

你也可以给方法传递一个 RowBounds 实例来限制查询结果。

映射器注解

因为最初设计时,MyBatis 是一个 XML 驱动的框架。配置信息是基于 XML 的,而且映射语句也是定义在 XML 中的。而到了 MyBatis 3,有新的可用的选择了。MyBatis 3 构建 在基于全面而且强大的 Java 配置 API 之上。这个配置 API 是基于 XML 的 MyBatis 配置的 基础,也是新的基于注解配置的基础。注解提供了一种简单的方式来实现简单映射语句,而 不会引入大量的开销。

注意 不幸的是,Java 注解限制了它们的表现和灵活。尽管很多时间都花调查,设计和实验上,最强大的 MyBatis 映射不能用注解来构建,那并不可笑。C#属性(做示例)就没 有这些限制,因此 MyBatis.NET 将会比 XML 有更丰富的选择。也就是说,基于 Java 注解 的配置离不开它的特性。

注解有下面这些:

注解	目标	相对应的 XML	
`@CacheNamespace`	`类`	`<cache>`	为给 存。 flushI
`@CacheNamespaceRef`	`类`	`<cacheRef>`	参照 用。 的字 。
`@ConstructorArgs`	`Method`	`<constructor>`	收集 构造 的数
`@Arg`	`方法`	`<arg>` `<idArg>`	单 独 Cons 性: id,col id 属

			属性
`@TypeDiscriminator`	方法	`<discriminator>`	一组实现。jdbcT cases
`@Case`	方法	`<case>`	单独性: va 是结 Resu 注解
`@Results`	方法	`<resultMap>`	结果果列情。Resu attrib mapp
`@Result`	方法	`<result>` `<id>`	在列射。javaT one,r 示了中的单独似,的,命名
`@One`	方法	`<association>`	复杂selec 法)的型的 API 注解 `fetc` the gl `lazy` mapp
			A ma of a c `sele` qualif stater can lo the a `fetc`

			the gl `lazy` mapp join m the A the lin that c refere
`@MapKey`	方法		复 杂 性 : s 器方 适类 在 Ja 为 Ja 用。
`@Options`	方法	映射语句的属性	这个 宽广 为 属 解变 晰的 性:us flush(result stater fetch useG keyP 是很 定"nu Optio 支配 免不
`@Insert` `@Update` `@Delete` `@Select`	方法	`<insert>` `<update>` `<delete>` `<select>`	这些 实 SC 数组 是字 的单 Java 空间" 欢迎 性:va 单独
			这些 个 类 运行

@InsertProvider @UpdateProvider @DeleteProvider @SelectProvider	方法	<insert> <update> <delete> <select>	运行 执行 这个 方法. 数对 只指 type, 限 定 法名 Selec 你以 态 SC
@Param	Parameter	N/A	如果 数, 这 方法 否则, 来被 参数) 等， @Pa 为 #{
@SelectKey	Method	<selectKey>	This < for m @Ins @Upc It is i you s anno any g via th confi state is the keyP prope that v value true SQL befor resu type sta
			This

`@ResultMap`	`Method`	N/A	elem `@Se` anno selec defin overr `@Co` both selec
`@ResultType`	`Method`	N/A	This a res returr have objec there @Re result the then neces anno @Se a res must @Re anno meth
`@Flush`	`Method`	N/A	If this callec `Sql` via m interf

映射申明样例

这个例子展示了如何使用 @SelectKey 注解来在插入前读取数据库序列的值：

```
@Insert("insert into table3 (id, name) values(#{nameId}, #{name}
)")
@SelectKey(statement="call next value for TestSequence", keyProp
erty="nameId", before=**true**, resultType=**int.class**)
**int** insertTable3(Name name);
```

这个例子展示了如何使用 @SelectKey 注解来在插入后读取数据库识别列的值：

```
@Insert("insert into table2 (name) values(#{name})")
@SelectKey(statement="call identity()", keyProperty="nameId", be
fore=**false**, resultType=**int.class**)
**int** insertTable2(Name name);
```

This example shows using the `@Flush` annotation to call the `SqlSession#flushStatements()`:

```
@Flush
List<BatchResult> flush();
```

These examples show how to name a ResultMap by specifying id attribute of @Results annotation.

```
@Results(id = "userResult", value = {
  @Result(property = "id", column = "uid", id = **true**),
  @Result(property = "firstName", column = "first_name"),
  @Result(property = "lastName", column = "last_name")
})
@Select("select * from users where id = #{id}")
User getUserById(Integer id);

@Results(id = "companyResults")
@ConstructorArgs({
  @Arg(property = "id", column = "cid", id = **true**),
  @Arg(property = "name", column = "name")
})
@Select("select * from company where id = #{id}")
Company getCompanyById(Integer id);
```

SQL语句构建器类

问题

Java程序员面对的最痛苦的事情之一就是在Java代码中嵌入SQL语句。这么来做通常是由于SQL语句需要动态来生成-否则可以将它们放到外部文件或者存储过程中。正如你已经看到的那样，MyBatis在它的XML映射特性中有一个强大的动态SQL生成方案。但有时在Java代码内部创建SQL语句也是必要的。此时，MyBatis有另外一个特性可以帮到你，在减少典型的加号,引号,新行,格式化问题和嵌入条件来处理多余的逗号或 AND 连接词之前。事实上，在Java代码中来动态生成SQL代码就是一场噩梦。例如：

```
String sql = "SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME,
"
"P.LAST_NAME,P.CREATED_ON, P.UPDATED_ON " +
"FROM PERSON P, ACCOUNT A " +
"INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT_ID " +
"INNER JOIN COMPANY C on D.COMPANY_ID = C.ID " +
"WHERE (P.ID = A.ID AND P.FIRST_NAME like ?) " +
"OR (P.LAST_NAME like ?) " +
"GROUP BY P.ID " +
"HAVING (P.LAST_NAME like ?) " +
"OR (P.FIRST_NAME like ?) " +
"ORDER BY P.ID, P.FULL_NAME";
```

The Solution

MyBatis 3提供了方便的工具类来帮助解决该问题。使用SQL类，简单地创建一个实例来调用方法生成SQL语句。上面示例中的问题就像重写SQL类那样：

```
private String selectPersonSql() {
  return new SQL() {{
    SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");
    SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");
    FROM("PERSON P");
    FROM("ACCOUNT A");
    INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");
    INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");
    WHERE("P.ID = A.ID");
    WHERE("P.FIRST_NAME like ?");
    OR();
    WHERE("P.LAST_NAME like ?");
    GROUP_BY("P.ID");
    HAVING("P.LAST_NAME like ?");
    OR();
    HAVING("P.FIRST_NAME like ?");
    ORDER_BY("P.ID");
    ORDER_BY("P.FULL_NAME");
  }}.toString();
}
```

该例中有什么特殊之处？当你仔细看时，那不用担心偶然间重复出现的"AND"关键字，或者在"WHERE"和"AND"之间的选择，抑或什么都不选。该SQL类非常注意"WHERE"应该出现在何处，哪里又应该使用"AND"，还有所有的字符串链接。

SQL类

这里给出一些示例：

```
// Anonymous inner class
public String deletePersonSql() {
  return new SQL() {{
    DELETE_FROM("PERSON");
    WHERE("ID = ${id}");
  }}.toString();
}

// Builder / Fluent style
public String insertPersonSql() {
  String sql = new SQL()
    .INSERT_INTO("PERSON")
    .VALUES("ID, FIRST_NAME", "${id}, ${firstName}")
    .VALUES("LAST_NAME", "${lastName}")
    .toString();
  return sql;
}

// With conditionals (note the final parameters, required for th
e anonymous inner class to access them)
```

```
public String selectPersonLike(final String id, final String fir
stName, final String lastName) {
  return new SQL() {{
    SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_N
AME");
    FROM("PERSON P");
    if (id != null) {
      WHERE("P.ID like ${id}");
    }
    if (firstName != null) {
      WHERE("P.FIRST_NAME like ${firstName}");
    }
    if (lastName != null) {
      WHERE("P.LAST_NAME like ${lastName}");
    }
    ORDER_BY("P.LAST_NAME");
  }}.toString();
}

public String deletePersonSql() {
  return new SQL() {{
    DELETE_FROM("PERSON");
    WHERE("ID = ${id}");
  }}.toString();
}

public String insertPersonSql() {
  return new SQL() {{
    INSERT_INTO("PERSON");
    VALUES("ID, FIRST_NAME", "${id}, ${firstName}");
    VALUES("LAST_NAME", "${lastName}");
  }}.toString();
}

public String updatePersonSql() {
  return new SQL() {{
    UPDATE("PERSON");
    SET("FIRST_NAME = ${firstName}");
    WHERE("ID = ${id}");
  }}.toString();
}
```

方法	描述
`SELECT(String)`	开始或插入到 `SELECT` 子句。 可以被多次调用，参数也会添加到 `SELECT` 子句。 参数通常使用逗号分隔的列名和别名列表，但也可以是数据库驱动程序接受的任意类型。
	开始或插入到 `SELECT` 子句， 也可以插入 `DISTINCT` 关键字到生成的查询语句中。

`SELECT_DISTINCT(String)`	可以被多次调用，参数也会添加到 `SELECT` 子句。 参数通常使用逗号分隔的列名和别名列表，但也可以是数据库驱动程序接受的任意类型。
`FROM(String)`	开始或插入到 `FROM` 子句。 可以被多次调用，参数也会添加到 `FROM` 子句。 参数通常是表名或别名，也可以是数据库驱动程序接受的任意类型。

`JOIN(String)`
`INNER_JOIN(String)`
`LEFT_OUTER_JOIN(String)`
`RIGHT_OUTER_JOIN(String)`
	基于调用的方法，添加新的合适类型的 `JOIN` 子句。 参数可以包含由列命和join on条件组合成标准的join。

`WHERE(String)`	插入新的 `WHERE` 子句条件， 由 `AND` 链接。可以多次被调用，每次都由 `AND` 来链接新条件。使用 `OR()` 来分隔 `OR` 。
`OR()`	使用 `OR` 来分隔当前的 `WHERE` 子句条件。 可以被多次调用，但在一行中多次调用或生成不稳定的 `SQL` 。
`AND()`	使用 `AND` 来分隔当前的 `WHERE` 子句条件。 可以被多次调用，但在一行中多次调用或生成不稳定的 `SQL` 。因为 `WHERE` 和 `HAVING` 二者都会自动链接 `AND` , 这是非常罕见的方法，只是为了完整性才被使用。
`GROUP_BY(String)`	插入新的 `GROUP BY` 子句元素，由逗号连接。 可以被多次调用，每次都由逗号连接新的条件。
`HAVING(String)`	插入新的 `HAVING` 子句条件。 由AND连接。可以被多次调用，每次都由 `AND` 来连接新的条件。使用 `OR()` 来分隔 `OR` .
`ORDER_BY(String)`	插入新的 `ORDER BY` 子句元素， 由逗号连接。可以多次被调用，每次由逗号连接新的条件。
`DELETE_FROM(String)`	开始一个delete语句并指定需要从哪个表删除的表名。通常它后面都会跟着WHERE语句！
`INSERT_INTO(String)`	开始一个insert语句并指定需要插入数据的表名。后面都会跟着一个或者多个VALUES()。
`SET(String)`	针对update语句，插入到"set"列表中
	开始一个update语句并指定需要更新的表

`UPDATE(String)`	明。后面都会跟着一个或者多个SET()，通常也会有一个WHERE()。
`VALUES(String, String)`	插入到insert语句中。第一个参数是要插入的列名，第二个参数则是该列的值。

SqlBuilder 和 SelectBuilder (已经废弃)

在3.2版本之前，我们使用了一点不同的做法，通过实现ThreadLocal变量来掩盖一些导致Java DSL麻烦的语言限制。但这种方式已经废弃了，现代的框架都欢迎人们使用构建器类型和匿名内部类的想法。因此，SelectBuilder 和 SqlBuilder 类都被废弃了。

下面的方法仅仅适用于废弃的SqlBuilder 和 SelectBuilder 类。

方法	描述
`BEGIN()` / `RESET()`	这些方法清空SelectBuilder类的ThreadLocal状态，并且准备一个新的构建语句。开始新的语句时， `BEGIN()` 读取得最好。 由于一些原因（在某些条件下，也许是逻辑需要一个完全不同的语句），在执行中清理语句 `RESET()` 读取得最好。
`SQL()`	返回生成的 `SQL()` 并重置 `SelectBuilder` 状态 (好像 `BEGIN()` 或 `RESET()` 被调用了). 因此，该方法只能被调用一次！

SelectBuilder 和 SqlBuilder 类并不神奇，但是知道它们如何工作也是很重要的。SelectBuilder 使用 SqlBuilder 使用了静态导入和ThreadLocal变量的组合来开启整洁语法，可以很容易地和条件交错。使用它们，静态导入类的方法即可，就像这样(一个或其它，并非两者):

```
import static org.apache.ibatis.jdbc.SelectBuilder.*;
```

```
import static org.apache.ibatis.jdbc.SqlBuilder.*;
```

这就允许像下面这样来创建方法：

```
/* DEPRECATED */
public String selectBlogsSql() {
  BEGIN(); // Clears ThreadLocal variable
  SELECT("*");
  FROM("BLOG");
  return SQL();
}
```

```
/* DEPRECATED */
private String selectPersonSql() {
  BEGIN(); // Clears ThreadLocal variable
  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");
  SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");
  FROM("PERSON P");
  FROM("ACCOUNT A");
  INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");
  INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");
  WHERE("P.ID = A.ID");
  WHERE("P.FIRST_NAME like ?");
  OR();
  WHERE("P.LAST_NAME like ?");
  GROUP_BY("P.ID");
  HAVING("P.LAST_NAME like ?");
  OR();
  HAVING("P.FIRST_NAME like ?");
  ORDER_BY("P.ID");
  ORDER_BY("P.FULL_NAME");
  return SQL();
}
```

Logging

Mybatis内置的日志工厂提供日志功能，具体的日志实现有以下几种工具：

- SLF4J
- Apache Commons Logging
- Log4j 2
- Log4j
- JDK logging

具体选择哪个日志实现工具由MyBatis的内置日志工厂确定。它会使用最先找到的（按上文列举的顺序查找）。 如果一个都未找到，日志功能就会被禁用。

不少应用服务器的classpath中已经包含Commons Logging，如Tomcat和WebShpere， 所以MyBatis会把它作为具体的日志实现。记住这点非常重要。这将意味着，在诸如 WebSphere的环境中——WebSphere提供了Commons Logging的私有实现，你的Log4J配置将被忽略。 这种做法不免让人悲催，MyBatis怎么能忽略你的配置呢？事实上，因Commons Logging已经存 在了，按照优先级顺序，Log4J自然就被忽略了！不过，如果你的应用部署在一个包含Commons Logging的环境， 而你又想用其他的日志框架，你可以根据需要调用如下的某一方法：

```
org.apache.ibatis.logging.LogFactory.useSlf4jLogging();
org.apache.ibatis.logging.LogFactory.useLog4JLogging();
org.apache.ibatis.logging.LogFactory.useJdkLogging();
org.apache.ibatis.logging.LogFactory.useCommonsLogging();
org.apache.ibatis.logging.LogFactory.useStdOutLogging();
```

如果的确需要调用以上的某个方法，请在其他所有MyBatis方法之前调用它。另外，只有在相应日志实现中存在 的前提下，调用对应的方法才是有意义的，否则MyBatis一概忽略。如你环境中并不存在Log4J，你却调用了 相应的方法，MyBatis就会忽略这一调用，代之默认的查找顺序查找日志实现。

关于SLF4J、Apache Commons Logging、Apache Log4J和JDK Logging的API介绍已经超出本文档的范围。 不过，下面的例子可以作为一个快速入门。关于这些日志框架的更多信息，可以参考以下链接：

- Apache Commons Logging
- Apache Log4j
- JDK Logging API

Logging Configuration

MyBatis可以对包、类、命名空间和全限定的语句记录日志。

具体怎么做，视使用的日志框架而定，这里以Log4J为例。配置日志功能非常简单：添加几个配置文件， 如log4j.properties,再添加个jar包，如log4j.jar。下面是具体的例子，共两个步骤：

步骤1： 添加 Log4J 的 jar 包

因为采用Log4J，要确保在应用中对应的jar包是可用的。要满足这一点，只要将jar包添加到应用的classpath中即可。 Log4J的jar包可以从上面的链接中下载。

具体而言，对于web或企业应用，需要将 `log4j.jar` 添加到 `WEB-INF/lib` 目录； 对于独立应用， 可以将它添加到jvm的 `-classpath` 启动参数中。

步骤2：配置Log4J

配置Log4J比较简单， 比如需要记录这个mapper接口的日志:

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

只要在应用的classpath中创建一个名称为 `log4j.properties` 的文件， 文件的具体内容如下：

```
# Global logging configuration
log4j.rootLogger=ERROR, stdout
# MyBatis logging configuration...
log4j.logger.org.mybatis.example.BlogMapper=TRACE
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

添加以上配置后，Log4J就会把 `org.mybatis.example.BlogMapper` 的详细执行日志记录下来，对于应用中的其它类则仅仅记录错误信息。

也可以将日志从整个mapper接口级别调整到到语句级别，从而实现更细粒度的控制。如下配置只记录 `selectBlog` 语句的日志：

```
log4j.logger.org.mybatis.example.BlogMapper.selectBlog=TRACE
```

与此相对，可以对一组mapper接口记录日志，只要对mapper接口所在的包开启日志功能即可：

```
log4j.logger.org.mybatis.example=TRACE
```

某些查询可能会返回大量的数据，只想记录其执行的SQL语句该怎么办？为此，Mybatis中SQL语 句的日志级别被设为DEBUG（JDK Logging中为FINE），结果日志的级别为TRACE（JDK Logging中为FINER)。所以，只要将日志级别调整为DEBUG即可达到目的：

```
log4j.logger.org.mybatis.example=DEBUG
```

要记录日志的是类似下面的mapper文件而不是mapper接口又该怎么呢？

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

对这个文件记录日志，只要对命名空间增加日志记录功能即可：

```
log4j.logger.org.mybatis.example.BlogMapper=TRACE
```

进一步，要记录具体语句的日志可以这样做：

```
log4j.logger.org.mybatis.example.BlogMapper.selectBlog=TRACE
```

看到了吧，两种配置没差别！

配置文件 `log4j.properties` 的余下内容是针对日志格式的，这一内容已经超出本 文档范围。关于Log4J的更多内容，可以参考Log4J的网站。不过，可以简单试一下看看，不同的配置 会产生什么不一样的效果。

项目文档

项目总体信息

本文档提供了与本项目相关信息的各种文档和链接的概述。这里的所有内容都是根据本项目信息生成的，自动生成过程依靠 Maven 。

概述

文档	描述
持续集成	持续集成是以一定的频率和固定的基础进行代码构建和测试的过程，这里列出了所有的持续集成过程。
项目依赖	这一文档列出了项目的依赖并提供了每个依赖的相关信息。
Dependency Information	This document describes how to to include this project as a dependency using various dependency management tools.
Distribution Management	This document provides informations on the distribution management of this project.
问题跟踪	这是本项目中的问题管理系统的链接。用户可以在这里进行问题(故障、特性、变更请求)的创建和查询。
项目授权	这是关于本项目授权的定义。
邮件列表	本文档提供了本项目的邮件列表的订阅和归档信息。
Plugin Management	This document lists the plugins that are defined through pluginManagement.
Project Plugins	This document lists the build plugins and the report plugins used by this project.
项目团队	本文档提供了本项目中成员的信息.他们是在本项目中以某种形式做出了贡献的个人.
源代码库	这是一个在线代码库,它可以通过Web浏览器进行查看.
项目概要	此文档列出了该项目其它的相关信息

概述

本项目使用 Travis CI。

访问

以下是在本项目中使用的持续集成系统。

```
[https://travis-ci.org/mybatis/mybatis-3/](https://travis-ci.org
/mybatis/mybatis-3/)
```

提醒方法

未定义任何的提醒方法，请以后再查看。

项目依赖

compile

以下是这一项目的编译依赖列表。编译和运行本应用需要这些依赖。

GroupId	ArtifactId	版本	类型	License	备注
cglib	cglib	3.1	jar	ASF 2.0	Yes
commons-logging	commons-logging	1.2	jar	The Apache Software License, Version 2.0	Yes
log4j	log4j	1.2.17	jar	The Apache Software License, Version 2.0	Yes
ognl	ognl	3.0.11	jar	The Apache Software License, Version 2.0	Yes
org.apache.logging.log4j	log4j-core	2.3	jar	The Apache Software License, Version 2.0	Yes
org.javassist	javassist	3.20.0-GA	jar	MPL 1.1-LGPL 2.1-Apache License 2.0	Yes
org.slf4j	slf4j-api	1.7.12	jar	MIT License	Yes
org.slf4j	slf4j-log4j12	1.7.12	jar	MIT License	Yes

test

以下是这一项目的测试依赖列表。这些依赖仅仅在编译和运行本应用中的单元测试时需要。

GroupId	ArtifactId	版本	类型	License
commons-dbcp	commons-dbcp	1.4	jar	The Apache Software License, Version 2.0
javax.transaction	transaction-api	1.1	jar	-
junit	junit	4.12	jar	Eclipse Public License 1.0
org.apache.derby	derby	10.11.1.1	jar	Apache 2
org.apache.velocity	velocity	1.7	jar	The Apache Software License, Version 2.0
org.hsqldb	hsqldb	2.3.2	jar	HSQLDB License, a BSD open source license
org.mockito	mockito-core	1.10.19	jar	The MIT License
postgresql	postgresql	9.1-901-1.jdbc4	jar	BSD License

Dependency Information

Apache Maven

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.4.0-SNAPSHOT</version>
</dependency>
```

Apache Buildr

```
'org.mybatis:mybatis:jar:3.4.0-SNAPSHOT'
```

Apache Ivy

```
<dependency org="org.mybatis" name="mybatis" rev="3.4.0-SNAPSHOT
">
  <artifact name="mybatis" type="jar" />
</dependency>
```

Groovy Grape

```
@Grapes(
@Grab(group='org.mybatis', module='mybatis', version='3.4.0-SNAP
SHOT')
)
```

Grails

```
compile 'org.mybatis:mybatis:3.4.0-SNAPSHOT'
```

Leiningen

```
[org.mybatis/mybatis "3.4.0-SNAPSHOT"]
```

SBT

```
libraryDependencies += "org.mybatis" % "mybatis" % "3.4.0-SNAPSH
OT"
```

Overview

The following is the distribution management information used by this project.

Repository - sonatype-nexus-staging

https://oss.sonatype.org/service/local/staging/deploy/maven2/

Snapshot Repository - sonatype-nexus-snapshots

https://oss.sonatype.org/content/repositories/snapshots/

Site - gh-pages

git:ssh://git@github.com/mybatis/mybatis-3.git?gh-pages#

概述

本项目使用GitHub Issue Management管理问题。

问题跟踪

问题、故障和新特性请求请通过下述的本项目的问题跟踪系统提交。

```
[https://github.com/mybatis/mybatis-3/issues](https://github.com
/mybatis/mybatis-3/issues)
```

概述

一般来说，项目中所列的授权仅对项目自身而言，不包括项目的依赖。

项目授权

The Apache Software License, Version 2.0

```
                              Apache License
                        Version 2.0, January 2004
                     http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1\. Definitions.

      "License" shall mean the terms and conditions for use, rep
roduction,
      and distribution as defined by Sections 1 through 9 of thi
s document.

      "Licensor" shall mean the copyright owner or entity author
ized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity a
nd all
      other entities that control, are controlled by, or are und
er common
      control with that entity. For the purposes of this definit
ion,
      "control" means (i) the power, direct or indirect, to caus
e the
      direction or management of such entity, whether by contrac
t or
      otherwise, or (ii) ownership of fifty percent (50%) or mor
e of the
      outstanding shares, or (iii) beneficial ownership of such
entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making mod
ifications,
      including but not limited to software source code, documen
tation
```

```
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanica
l
      transformation or translation of a Source form, including
but
      not limited to compiled object code, generated documentati
on,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Sourc
e or
      Object form, made available under the License, as indicate
d by a
      copyright notice that is included in or attached to the wo
rk
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source
or Object
      form, that is based on (or derived from) the Work and for
which the
      editorial revisions, annotations, elaborations, or other m
odifications
      represent, as a whole, an original work of authorship. For
 the purposes
      of this License, Derivative Works shall not include works
that remain
      separable from, or merely link (or bind by name) to the in
terfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, includin
g
      the original version of the Work and any modifications or
additions
      to that Work or Derivative Works thereof, that is intentio
nally
      submitted to Licensor for inclusion in the Work by the cop
yright owner
      or by an individual or Legal Entity authorized to submit o
n behalf of
      the copyright owner. For the purposes of this definition,
"submitted"
      means any form of electronic, verbal, or written communica
tion sent
      to the Licensor or its representatives, including but not
limited to
      communication on electronic mailing lists, source code con
trol systems,
      and issue tracking systems that are managed by, or on beha
lf of, the
      Licensor for the purpose of discussing and improving the W
```

```
ork, but
      excluding communication that is conspicuously marked or ot
herwise
      designated in writing by the copyright owner as "Not a Con
tribution."

      "Contributor" shall mean Licensor and any individual or Le
gal Entity
      on behalf of whom a Contribution has been received by Lice
nsor and
      subsequently incorporated within the Work.

   2\. Grant of Copyright License. Subject to the terms and cond
itions of
      this License, each Contributor hereby grants to You a perp
etual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
      copyright license to reproduce, prepare Derivative Works o
f,
      publicly display, publicly perform, sublicense, and distri
bute the
      Work and such Derivative Works in Source or Object form.

   3\. Grant of Patent License. Subject to the terms and conditi
ons of
      this License, each Contributor hereby grants to You a perp
etual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
      (except as stated in this section) patent license to make,
 have made,
      use, offer to sell, sell, import, and otherwise transfer t
he Work,
      where such license applies only to those patent claims lic
ensable
      by such Contributor that are necessarily infringed by thei
r
      Contribution(s) alone or by combination of their Contribut
ion(s)
      with the Work to which such Contribution(s) was submitted.
 If You
      institute patent litigation against any entity (including
a
      cross-claim or counterclaim in a lawsuit) alleging that th
e Work
      or a Contribution incorporated within the Work constitutes
 direct
      or contributory patent infringement, then any patent licen
ses
      granted to You under this License for that Work shall term
inate
      as of the date such litigation is filed.
```

```
   4\. Redistribution. You may reproduce and distribute copies o
f the
      Work or Derivative Works thereof in any medium, with or wi
thout
      modifications, and in Source or Object form, provided that
 You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent n
otices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative 
Works
          that You distribute, all copyright, patent, trademark,
 and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any par
t of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of i
ts
          distribution, then any Derivative Works that You distr
ibute must
          include a readable copy of the attribution notices con
tained
          within such NOTICE file, excluding those notices that 
do not
          pertain to any part of the Derivative Works, in at lea
st one
          of the following places: within a NOTICE text file dis
tributed
          as part of the Derivative Works; within the Source for
m or
          documentation, if provided along with the Derivative W
orks; or,
          within a display generated by the Derivative Works, if
 and
          wherever such third-party notices normally appear. The
 contents
          of the NOTICE file are for informational purposes only
 and
          do not modify the License. You may add Your own attrib
ution
          notices within Derivative Works that You distribute, a
longside
          or as an addendum to the NOTICE text from the Work, pr
ovided
```

```
          that such additional attribution notices cannot be con
strued
          as modifying the License.

      You may add Your own copyright statement to Your modificat
ions and
      may provide additional or different license terms and cond
itions
      for use, reproduction, or distribution of Your modificatio
ns, or
      for any such Derivative Works as a whole, provided Your us
e,
      reproduction, and distribution of the Work otherwise compl
ies with
      the conditions stated in this License.

   5\. Submission of Contributions. Unless You explicitly state
otherwise,
      any Contribution intentionally submitted for inclusion in
the Work
      by You to the Licensor shall be under the terms and condit
ions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede
or modify
      the terms of any separate license agreement you may have e
xecuted
      with Licensor regarding such Contributions.

   6\. Trademarks. This License does not grant permission to use
 the trade
      names, trademarks, service marks, or product names of the
Licensor,
      except as required for reasonable and customary use in des
cribing the
      origin of the Work and reproducing the content of the NOTI
CE file.

   7\. Disclaimer of Warranty. Unless required by applicable law
 or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASI
S,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either expre
ss or
      implied, including, without limitation, any warranties or
conditions
      of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FO
R A
      PARTICULAR PURPOSE. You are solely responsible for determi
ning the
      appropriateness of using or redistributing the Work and as
sume any
```

```
      risks associated with Your exercise of permissions under t
his License.

   8\. Limitation of Liability. In no event and under no legal t
heory,
      whether in tort (including negligence), contract, or other
wise,
      unless required by applicable law (such as deliberate and 
grossly
      negligent acts) or agreed to in writing, shall any Contrib
utor be
      liable to You for damages, including any direct, indirect,
 special,
      incidental, or consequential damages of any character aris
ing as a
      result of this License or out of the use or inability to u
se the
      Work (including but not limited to damages for loss of goo
dwill,
      work stoppage, computer failure or malfunction, or any and
 all
      other commercial damages or losses), even if such Contribu
tor
      has been advised of the possibility of such damages.

   9\. Accepting Warranty or Additional Liability. While redistr
ibuting
      the Work or Derivative Works thereof, You may choose to of
fer,
      and charge a fee for, acceptance of support, warranty, ind
emnity,
      or other liability obligations and/or rights consistent wi
th this
      License. However, in accepting such obligations, You may a
ct only
      on Your own behalf and on Your sole responsibility, not on
 behalf
      of any other Contributor, and only if You agree to indemni
fy,
      defend, and hold each Contributor harmless for any liabili
ty
      incurred by, or claims asserted against, such Contributor 
by reason
      of your accepting any such warranty or additional liabilit
y.

   END OF TERMS AND CONDITIONS

   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the follo
wing
      boilerplate notice, with the fields enclosed by brackets "
```

```
[]"
      replaced with your own identifying information. (Don't inc
lude
      the brackets!)  The text should be enclosed in the appropr
iate
      comment syntax for the file format. We also recommend that
 a
      file or class name and description of purpose be included
on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.

   Copyright [yyyy] [name of copyright owner]

   Licensed under the Apache License, Version 2.0 (the "License"
);
   you may not use this file except in compliance with the Licen
se.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, so
ftware
   distributed under the License is distributed on an "AS IS" BA
SIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
   See the License for the specific language governing permissio
ns and
   limitations under the License.
```

The Apache Software License, Version 2.0

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1\. Definitions.

      "License" shall mean the terms and conditions for use, rep
roduction,
      and distribution as defined by Sections 1 through 9 of thi
s document.

      "Licensor" shall mean the copyright owner or entity author
ized by
      the copyright owner that is granting the License.
```

```
      "Legal Entity" shall mean the union of the acting entity a
nd all
      other entities that control, are controlled by, or are und
er common
      control with that entity. For the purposes of this definit
ion,
      "control" means (i) the power, direct or indirect, to caus
e the
      direction or management of such entity, whether by contrac
t or
      otherwise, or (ii) ownership of fifty percent (50%) or mor
e of the
      outstanding shares, or (iii) beneficial ownership of such
entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making mod
ifications,
      including but not limited to software source code, documen
tation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanica
l
      transformation or translation of a Source form, including
but
      not limited to compiled object code, generated documentati
on,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Sourc
e or
      Object form, made available under the License, as indicate
d by a
      copyright notice that is included in or attached to the wo
rk
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source
or Object
      form, that is based on (or derived from) the Work and for
which the
      editorial revisions, annotations, elaborations, or other m
odifications
      represent, as a whole, an original work of authorship. For
 the purposes
      of this License, Derivative Works shall not include works
that remain
      separable from, or merely link (or bind by name) to the in
terfaces of,
      the Work and Derivative Works thereof.
```

```
      "Contribution" shall mean any work of authorship, includin
g
      the original version of the Work and any modifications or
additions
      to that Work or Derivative Works thereof, that is intentio
nally
      submitted to Licensor for inclusion in the Work by the cop
yright owner
      or by an individual or Legal Entity authorized to submit o
n behalf of
      the copyright owner. For the purposes of this definition,
"submitted"
      means any form of electronic, verbal, or written communica
tion sent
      to the Licensor or its representatives, including but not
limited to
      communication on electronic mailing lists, source code con
trol systems,
      and issue tracking systems that are managed by, or on beha
lf of, the
      Licensor for the purpose of discussing and improving the W
ork, but
      excluding communication that is conspicuously marked or ot
herwise
      designated in writing by the copyright owner as "Not a Con
tribution."

      "Contributor" shall mean Licensor and any individual or Le
gal Entity
      on behalf of whom a Contribution has been received by Lice
nsor and
      subsequently incorporated within the Work.

   2\. Grant of Copyright License. Subject to the terms and cond
itions of
      this License, each Contributor hereby grants to You a perp
etual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
      copyright license to reproduce, prepare Derivative Works o
f,
      publicly display, publicly perform, sublicense, and distri
bute the
      Work and such Derivative Works in Source or Object form.

   3\. Grant of Patent License. Subject to the terms and conditi
ons of
      this License, each Contributor hereby grants to You a perp
etual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevoc
able
      (except as stated in this section) patent license to make,
```

```
 have made,
      use, offer to sell, sell, import, and otherwise transfer t
he Work,
      where such license applies only to those patent claims lic
ensable
      by such Contributor that are necessarily infringed by thei
r
      Contribution(s) alone or by combination of their Contribut
ion(s)
      with the Work to which such Contribution(s) was submitted.
 If You
      institute patent litigation against any entity (including
a
      cross-claim or counterclaim in a lawsuit) alleging that th
e Work
      or a Contribution incorporated within the Work constitutes
 direct
      or contributory patent infringement, then any patent licen
ses
      granted to You under this License for that Work shall term
inate
      as of the date such litigation is filed.

   4\. Redistribution. You may reproduce and distribute copies o
f the
      Work or Derivative Works thereof in any medium, with or wi
thout
      modifications, and in Source or Object form, provided that
 You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent n
otices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative
Works
          that You distribute, all copyright, patent, trademark,
 and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any par
t of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of i
ts
          distribution, then any Derivative Works that You distr
ibute must
          include a readable copy of the attribution notices con
tained
```

```
          within such NOTICE file, excluding those notices that 
do not
          pertain to any part of the Derivative Works, in at lea
st one
          of the following places: within a NOTICE text file dis
tributed
          as part of the Derivative Works; within the Source for
m or
          documentation, if provided along with the Derivative W
orks; or,
          within a display generated by the Derivative Works, if
 and
          wherever such third-party notices normally appear. The
 contents
          of the NOTICE file are for informational purposes only
 and
          do not modify the License. You may add Your own attrib
ution
          notices within Derivative Works that You distribute, a
longside
          or as an addendum to the NOTICE text from the Work, pr
ovided
          that such additional attribution notices cannot be con
strued
          as modifying the License.

      You may add Your own copyright statement to Your modificat
ions and
      may provide additional or different license terms and cond
itions
      for use, reproduction, or distribution of Your modificatio
ns, or
      for any such Derivative Works as a whole, provided Your us
e,
      reproduction, and distribution of the Work otherwise compl
ies with
      the conditions stated in this License.

   5\. Submission of Contributions. Unless You explicitly state 
otherwise,
      any Contribution intentionally submitted for inclusion in 
the Work
      by You to the Licensor shall be under the terms and condit
ions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede 
or modify
      the terms of any separate license agreement you may have e
xecuted
      with Licensor regarding such Contributions.

   6\. Trademarks. This License does not grant permission to use
 the trade
```

```
      names, trademarks, service marks, or product names of the
Licensor,
      except as required for reasonable and customary use in des
cribing the
      origin of the Work and reproducing the content of the NOTI
CE file.

   7\. Disclaimer of Warranty. Unless required by applicable law
 or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASI
S,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either expre
ss or
      implied, including, without limitation, any warranties or
conditions
      of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FO
R A
      PARTICULAR PURPOSE. You are solely responsible for determi
ning the
      appropriateness of using or redistributing the Work and as
sume any
      risks associated with Your exercise of permissions under t
his License.

   8\. Limitation of Liability. In no event and under no legal t
heory,
      whether in tort (including negligence), contract, or other
wise,
      unless required by applicable law (such as deliberate and
grossly
      negligent acts) or agreed to in writing, shall any Contrib
utor be
      liable to You for damages, including any direct, indirect,
 special,
      incidental, or consequential damages of any character aris
ing as a
      result of this License or out of the use or inability to u
se the
      Work (including but not limited to damages for loss of goo
dwill,
      work stoppage, computer failure or malfunction, or any and
 all
      other commercial damages or losses), even if such Contribu
tor
      has been advised of the possibility of such damages.

   9\. Accepting Warranty or Additional Liability. While redistr
ibuting
      the Work or Derivative Works thereof, You may choose to of
fer,
      and charge a fee for, acceptance of support, warranty, ind
emnity,
```

```
      or other liability obligations and/or rights consistent wi
th this
      License. However, in accepting such obligations, You may a
ct only
      on Your own behalf and on Your sole responsibility, not on
 behalf
      of any other Contributor, and only if You agree to indemni
fy,
      defend, and hold each Contributor harmless for any liabili
ty
      incurred by, or claims asserted against, such Contributor
by reason
      of your accepting any such warranty or additional liabilit
y.

   END OF TERMS AND CONDITIONS

   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the follo
wing
      boilerplate notice, with the fields enclosed by brackets "
[]"
      replaced with your own identifying information. (Don't inc
lude
      the brackets!)  The text should be enclosed in the appropr
iate
      comment syntax for the file format. We also recommend that
 a
      file or class name and description of purpose be included
on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.

   Copyright [yyyy] [name of copyright owner]

   Licensed under the Apache License, Version 2.0 (the "License"
);
   you may not use this file except in compliance with the Licen
se.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, so
ftware
   distributed under the License is distributed on an "AS IS" BA
SIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
   See the License for the specific language governing permissio
ns and
   limitations under the License.
```

项目邮件列表

这些是本项目建立的邮件列表。每个列表都有一个订阅、退订和归档的链接。

名称	订阅	取消订阅	发送邮件	归档	其他归档
mybatis-dev	订阅	取消订阅	发送邮件	groups.google.com	-
mybatis-user	订阅	取消订阅	发送邮件	groups.google.com	mybatis-user.963551.n3.nabble.com
mybatis-commits	订阅	取消订阅	发送邮件	groups.google.com	-

Project Plugin Management

GroupId	ArtifactId	Version
com.mycila	license-maven-plugin	2.11
org.apache.maven.plugins	maven-antrun-plugin	1.8
org.apache.maven.plugins	maven-assembly-plugin	2.5.5
org.apache.maven.plugins	maven-clean-plugin	2.6.1
org.apache.maven.plugins	maven-compiler-plugin	3.3
org.apache.maven.plugins	maven-dependency-plugin	2.10
org.apache.maven.plugins	maven-deploy-plugin	2.8.2
org.apache.maven.plugins	maven-install-plugin	2.5.2
org.apache.maven.plugins	maven-pdf-plugin	1.3
org.apache.maven.plugins	maven-release-plugin	2.5.2
org.apache.maven.plugins	maven-resources-plugin	2.7
org.apache.maven.plugins	maven-shade-plugin	2.4
org.apache.maven.plugins	maven-site-plugin	3.4
org.apache.maven.plugins	maven-surefire-plugin	2.18.1
org.codehaus.mojo	versions-maven-plugin	2.2
org.jacoco	http://jacoco-maven-plugin	0.7.5.201505241946

Project Build Plugins

GroupId	ArtifactId	Version
com.mycila	license-maven-plugin	2.11
org.apache.felix	maven-bundle-plugin	2.5.4
org.apache.maven.plugins	maven-clean-plugin	2.6.1
org.apache.maven.plugins	maven-compiler-plugin	3.3
org.apache.maven.plugins	maven-deploy-plugin	2.8.2
org.apache.maven.plugins	maven-enforcer-plugin	1.4
org.apache.maven.plugins	maven-install-plugin	2.5.2
org.apache.maven.plugins	maven-jar-plugin	2.6
org.apache.maven.plugins	maven-pdf-plugin	1.3
org.apache.maven.plugins	maven-resources-plugin	2.7
org.apache.maven.plugins	maven-scm-plugin	1.9.4
org.apache.maven.plugins	maven-shade-plugin	2.4
org.apache.maven.plugins	maven-site-plugin	3.4
org.apache.maven.plugins	maven-surefire-plugin	2.18.1
org.codehaus.mojo	animal-sniffer-maven-plugin	1.14
org.codehaus.mojo	clirr-maven-plugin	2.6.1
org.jacoco	http://jacoco-maven-plugin	0.7.5.201505241946

Project Report Plugins

GroupId	ArtifactId	Version
org.apache.maven.plugins	maven-changes-plugin	2.11
org.apache.maven.plugins	maven-javadoc-plugin	2.10.3
org.apache.maven.plugins	maven-jxr-plugin	2.5
org.apache.maven.plugins	maven-pmd-plugin	3.4
org.apache.maven.plugins	maven-project-info-reports-plugin	2.8
org.apache.maven.plugins	maven-surefire-report-plugin	2.18.1
org.codehaus.mojo	clirr-maven-plugin	2.6.1
org.codehaus.mojo	findbugs-maven-plugin	3.0.1
org.codehaus.mojo	jdepend-maven-plugin	2.0
org.codehaus.mojo	taglist-maven-plugin	2.4
org.codehaus.mojo	versions-maven-plugin	2.2
org.jacoco	http://jacoco-maven-plugin	0.7.5.201505241946

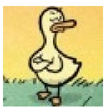
团队

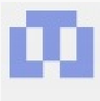
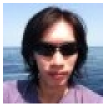
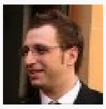
一个成功的项目要求许多人扮演许多的角色。其中一些成员编写代码或者文档，同时其他成员则通过测试、打补丁以及提建议等方式实现自己的价值。

一个团队由团队成员以及贡献者组成。团队成员直接访问项目的源代码并积极地参与编码工作。贡献者则通过向成员提交补丁和提出建议来完善项目。一个项目中贡献者的数量是不限的。请立刻加入到贡献行列中来吧。衷心感谢所有对项目做出贡献的人。

成员

以下是开发者的列表，他们有提交的权限，在项目中以某种方式直接做出了贡献。

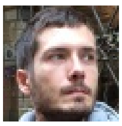
Image	Id	姓名	电子邮件
	cbegin	Clinton Begin	clinton.begin@gmail.com
	brandon.goodin	Brandon Goodin	brandon.goodin@gmail.co
	mentat6114	Christian Poitras	christian.poitras@ircm.qc.(
	emacarron	Eduardo Macarron	eduardo.macarron@gmail.
	mnesarco	Frank Martinez	mnesarco@gmail.com
	agustafson	Andrew Gustafson	gus4000@gmail.com
	hpresnall	Hunter Presnall	hpresnall@gmail.com
	harawata	Iwao Ave	harawata@gmail.com
	jeffgbutler	Jeff Butler	jeffgbutler@gmail.com

	hazendaz	Jeremy Landis	jeremylandis@hotmail.com
	nospam@kaigrabfelder.de	Kai Grabfelder	nospam@kaigrabfelder.de
	lmeadors	Larry Meadors	larry.meadors@gmail.com
	marcosperanza	Marco Speranza	marco.speranza79@gmail
	nmaves	Nathan Maves	nathan.maves@gmail.com
	putthibongb	Putthibong Boonbong	putthibongb@gmail.com
	simonetripodi	Simone Tripodi	simone.tripodi@gmail.com
	h3adache	Tim Chen	chengt@gmail.com

贡献者

以下的其他人通过建议，补丁或者文档对本项目做出了贡献。

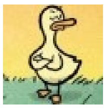
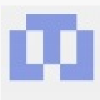
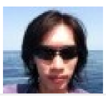
Image	姓名	电子邮件
	Adam Gent	adam.gent@evocatus.com
	Andrea Selva	selva.andre@gmail.com
	Antonio Sánchez	juntandolineas@gmail.com
	Arkadi Shishlov	arkadi.shishlov@gmail.com

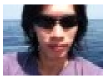
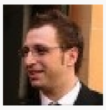
	Axel Doerfler	axel.doerfler@gmail.com
	Chris Dadej	chris.dadej@gmail.com
	Denis Vygovskiy	qizant@gmail.com
	Franta Mejta	mejta@rewor.cz
	Jurriaan Pruys	jurriaan@pruys.com
	Keith Wong	wongkwl@gmail.com
	Lasse Voss	lasse.voss@motor-talk-gmbh.de
	Luke Stevens	nosuchluke@gmail.com
	Paul Krause	paulkrause88@alum.mit.edu
	Peter Leibiger	kuhnroyal@gmail.com
	Riccardo Cossu	riccardo.cossu@gmail.com
	Tomá? Neuberg	neuberg@m-atelier.cz

成员

以下是开发者的列表，他们有提交的权限，在项目中以某种方式直接做出了贡献。

Image	Id	姓名	电子邮件

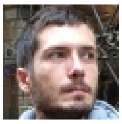
Image	Id	姓名	电子邮件
	cbegin	Clinton Begin	clinton.begin@gmail.com
	brandon.goodin	Brandon Goodin	brandon.goodin@gmail.co
	mentat6114	Christian Poitras	christian.poitras@ircm.qc.c
	emacarron	Eduardo Macarron	eduardo.macarron@gmail.
	mnesarco	Frank Martinez	mnesarco@gmail.com
	agustafson	Andrew Gustafson	gus4000@gmail.com
	hpresnall	Hunter Presnall	hpresnall@gmail.com
	harawata	Iwao Ave	harawata@gmail.com
	jeffgbutler	Jeff Butler	jeffgbutler@gmail.com
	hazendaz	Jeremy Landis	jeremylandis@hotmail.com
	nospam@kaigrabfelder.de	Kai Grabfelder	nospam@kaigrabfelder.de
	lmeadors	Larry Meadors	larry.meadors@gmail.com
	marcosperanza	Marco Speranza	marco.speranza79@gmail
	nmaves	Nathan Maves	nathan.maves@gmail.com
	putthibongb	Putthibong Boonbong	putthibongb@gmail.com

	[illegible]	[illegible]	[illegible]
	simonetripodi	Simone Tripodi	simone.tripodi@gmail.com
	h3adache	Tim Chen	chengt@gmail.com

贡献者

以下的其他人通过建议，补丁或者文档对本项目做出了贡献。

Image	姓名	电子邮件
	Adam Gent	adam.gent@evocatus.com
	Andrea Selva	selva.andre@gmail.com
	Antonio Sánchez	juntandolineas@gmail.com
	Arkadi Shishlov	arkadi.shishlov@gmail.com
	Axel Doerfler	axel.doerfler@gmail.com
	Chris Dadej	chris.dadej@gmail.com
	Denis Vygovskiy	qizant@gmail.com
	Franta Mejta	mejta@rewor.cz
	Jurriaan Pruys	jurriaan@pruys.com

	Lasse Voss	lasse.voss@motor-talk-gmbh.de
	Luke Stevens	nosuchluke@gmail.com
	Paul Krause	paulkrause88@alum.mit.edu
	Peter Leibiger	kuhnroyal@gmail.com
	Riccardo Cossu	riccardo.cossu@gmail.com
	Tomá? Neuberg	neuberg@m-atelier.cz

概述

This project uses Git to manage its source code. Instructions on Git use can be found at http://git-scm.com/documentation.

Web访问

以下是在线源代码库的链接。

```
[http://github.com/mybatis/mybatis-3](http://github.com/mybatis/mybatis-3)
```

匿名访问

The source can be checked out anonymously from Git with this command (See http://git-scm.com/docs/git-clone):

```
$ git clone ssh://github.com/mybatis/mybatis-3.git
```

开发者访问

Only project developers can access the Git tree via this method (See http://git-scm.com/docs/git-clone).

```
$ git clone ssh://git@github.com/mybatis/mybatis-3.git
```

通过防火墙访问

请参考使用的源代码管理工具的文档以获得更多的通过防火墙进行访问的信息。

项目概要

项目信息

字段	值
名称	mybatis
描述	The MyBatis data mapper framework makes it easier to use a relational database with object-oriented applications. MyBatis couples objects with stored procedures or SQL statements using a XML descriptor or annotations. Simplicity is the biggest advantage of the MyBatis data mapper over object relational mapping tools.
主页	http://www.mybatis.org/core/

项目组织

字段	值
名称	MyBatis.org
URL	http://www.mybatis.org/

构建信息

字段	值
GroupId	org.mybatis
ArtifactId	mybatis
版本	3.4.0-SNAPSHOT
类型	jar
Java Version	1.6

项目信息

字段	值
名称	mybatis
描述	The MyBatis data mapper framework makes it easier to use a relational database with object-oriented applications. MyBatis couples objects with stored procedures or SQL statements using a XML descriptor or annotations. Simplicity is the biggest advantage of the MyBatis data mapper over object relational mapping tools.
主页	http://www.mybatis.org/core/

项目组织

字段	值
名称	MyBatis.org
URL	http://www.mybatis.org/

构建信息

字段	值
GroupId	org.mybatis
ArtifactId	mybatis
版本	3.4.0-SNAPSHOT
类型	jar
Java Version	1.6

生成报表

本文档提供了对项目中各种报表的概述，这些报表生成自 Maven 每一个报表在以下简要介绍。

概述

文档	描述
JavaDocs	JavaDoc API documentation.
Source Xref	HTML based, cross-reference version of Java source code.
Test Source Xref	HTML based, cross-reference version of Java test source code.
JDepend	JDepend traverses Java class file directories and generates design quality metrics for each Java package. JDepend allows you to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively.
FindBugs	Generates a source code report with the FindBugs Library.
Surefire Report	Report on the test results of the project.
CPD	Duplicate code detection.
PMD	Verification of coding rules.
Tag List	Report on various tags found in the code.
Clirr	Report on binary and source API differences between releases
Dependency Updates Report	Provides details of the dependencies which have updated versions available.
Plugin Updates Report	Provides details of the plugins used by this project which have newer versions available.
Property Updates Report	Provides details of properties which control versions of dependencies and/or plugins, and indicates any newer versions which are available.

概述

文档	描述
JavaDocs	JavaDoc API documentation.
Source Xref	HTML based, cross-reference version of Java source code.
Test Source Xref	HTML based, cross-reference version of Java test source code.
JDepend	JDepend traverses Java class file directories and generates design quality metrics for each Java package. JDepend allows you to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively.
FindBugs	Generates a source code report with the FindBugs Library.
Surefire Report	Report on the test results of the project.
CPD	Duplicate code detection.
PMD	Verification of coding rules.
Tag List	Report on various tags found in the code.
Clirr	Report on binary and source API differences between releases
Dependency Updates Report	Provides details of the dependencies which have updated versions available.
Plugin Updates Report	Provides details of the plugins used by this project which have newer versions available.
Property Updates Report	Provides details of properties which control versions of dependencies and/or plugins, and indicates any newer versions which are available.

MyBatis 用户手册

MyBatis Generator介绍

MyBatis Generator (MBG) 是一个Mybatis的代码生成器 MyBatis 和 iBATIS. 他可以生成Mybatis各个版本的代码，和iBATIS 2.2.0版本以后的代码。 他可以内省数据库的表（或多个表）然后生成可以用来访问（多个）表的基础对象。 这样和数据库表进行交互时不需要创建对象和配置文件。 MBG的解决了对数据库操作有最大影响的一些简单的CRUD（插入，查询，更新，删除）操作。 您仍然需要对联合查询和存储过程手写SQL和对象。

MyBatis Generator 会生成:

- 匹配表结构的Java POJO，可能包括:
    - 一个和表主键匹配的类(如果存在主键[注:只有联合主键会有])
    - 一个包含了非主键字段的类(BLOB字段除外[注:单字段做主键时这里会包含])
    - 一个包含了BLOB字段的类 (如果表包含了BLOB字段)
    - 一个允许动态查询、更新和删除的类[注:指的是Example查询]

  这些类之间会有适当的继承关系。 请注意可以配置生成器来生成不同类型的POJO 的层次结构。 例如，如果您愿意您可能会选择针对每个表生成一个单独的实体对象。

- MyBatis/iBATIS 兼容 SQL 映射 XML 文件。MBG 在配置中为每个表简单的CRUD 操作生成 SQL。 生成的 SQL 语句包括：
    - insert (插入)
    - update by primary key (根据主键更新记录)
    - update by example (根据条件更新记录)
    - delete by primary key (根据主键删除记录)
    - delete by example (根据条件删除记录)
    - select by primary key (根据主键查询记录)
    - select by example (根据条件查询记录集)
    - count by example (根据条件查询记录总数)

  根据表的结构，生成的这些语句会有不同的变化（例如，如果表中没有主键，那么 MBG 将不会生成update by primary key方法）。

- Java客户端类会适当的使用上面的对象，生成Java客户端类时可选的。 MBG会为MyBatis 3.x生成如下客户端类：
    - 一个可以和MyBatis 3.x一起使用的mapper接口类MBG会为iBATIS 2.x生成如下的客户端类：
    - 符合 Spring 框架的DAO类。
    - 只使用iBATIS SQL映射API的DAO。 这种DAO可以通过下面两种方式生成：提供 `SqlMapClient` 通过构造方法或者setter注入。
    - 符合iBATIS DAO 框架的DAO (iBATIS可选的一部分，这一框架已经过时，我们建议您使用Spring框架替代)。

MyBatis generator 可以在迭代开发环境中良好的运行， 在持续的构建环境中作为一个ant任务或maven插件。 运行MBG时要记住以下重要的事：

1. MBG 会自动合并已经存在并且和新生成的文件重名的 XML。MBG 不会覆盖您对已经生成xml所做的修改。 您可以反复的运行而不必担心失去您自定义的更改。 MBG 将取代所有以前运行中生成的 XML 元素。
2. MBG 不会 合并 Java 文件,他可以覆盖已经存在的文件或者保存新生成的文件为一个不同的唯一的名字。 您可以手动合并这些更改。 当您使用Eclipse 插件时, MBG 可以自动合并 Java 文件.

依赖项

MBG依赖JRE,需要JRE6.0或以上版本。 此外，还有一个继承了DatabaseMetaData接口的JDBC驱动。 特别是 `getColumns` 和 `getPrimaryKeys` 两个方法时必须的。

支持

通过Mybatis用户邮件列表提供对Mybatis generator的支持。 您可以通过谷歌订阅或查看邮件列表的代码:

> http://groups.google.com/group/mybatis-user

如果您认为您找到了一个 bug，在您创建新的Issue前，请在用户列表中询问一下。如果您发现一个 bug，或者有新的功能要求，您可以在github上打开一个新的Issue：

> https://github.com/mybatis/generator/issues

译者

译者水平有限，如果您发现翻译不通顺的地方，您可以在下面的地址创建Issue：

> http://git.oschina.net/free/mybatis-generator-core/issues

译者列表

- abel533(isea533)
  - github:https://github.com/abel533
  - gitosc:http://git.oschina.net/free
  - blog:http://blog.csdn.net/isea533
- hs598375774sa
  - oschina:http://my.oschina.net/hssa
- '╋锦色乂年华2o1?

MyBatis Generator新增功能

Version 1.3.3

公告

- MyBatis Generator 需要 JRE 1.6 或更高版本的支持

改进

- 从 MyBatis 3.2 开始使用了新的 SQL 生成器类。 如果您必须在 MyBatis 旧版本上运行，在您的 < JavaClientGenerator > 配置上设置属性"useLegacyBuilder"为"true"。
- Issue #11 - maven插件现在会输出到maven日志 (thanks to Paul Krause)
- Issue #4 - 如果任何列将解析为一个 Java 保留字，生成器现在会抛出一个警告

Bugs 修复

- Fixed issue #648 - Missing imports when using constructor based models
- Issues #10 - EqualsHashCodePlugin now uses the java.util.Arrays methods for array attributes.

Version 1.3.2

Bugs 修复

- Add comments to generated constructors in the model classes so they can be merged.
- Support package names with Uppercase elements.
- Fixed issue #288 - Incorrect annotated countByExample method
- Fixed the Maven plugin so <properties> files can be found in the project classpath.
- Fixed issue #359 - make JdbcTypeInformation public
- Fixed Context.toXmlElement() method to include missing attributes
- Fixed CaseInsensitiveLikePlugin to add new methods to GeneratedCriteria inner class
- Issue #412 - update the documentation to reflect the difference in MyBatis3 regarding generated keys.
- Issue #440 - incorrect code generated for primitives with type handlers
- Issue #439 - use auto boxing for primitives when appropriate
- Issue #438 - keep primary key properties in database sequence
- Issue #507 - RowBounds plugin generates duplicate statements

- Issue #593 - CaseInsensitiveLikePlugin skipped Jdbc4 National Character Types

改进

- Added a new target runtime - MyBatis3Simple - that can be used to generate very simple CRUD operations on tables. This runtime produces far simpler MyBatis code than the normal MyBatis3 runtime. The produced code also has lower functionality that the normal MyBatis3 runtime. If you do not regularly use the "by example" methods, then the code produced by the MyBatis3Siple runtime may be more suitable to your project.
- Added a new plugin - VirtualPrimaryKey plugin - that can be used to specify columns that act as a primary key, even if they are not defined as primary keys in the database.
- Issue #328 - Added a new plugin - RowBounds plugin - that will generate an additional version of `selectByExample` that supports the MyBatis RowBounds function.
- Created a new reference page for supplied plugins: Supplied Plugins
- Issue #368 - Allowed for generation of Java Model only. If no SQLMapGenerator or JavaClientGenerator is specified in a context, then only a Java Model is generated. Also, if property "modelOnly" is set to "true" on a <table> element, then only model object and, possibly XML result maps, will be generated.
- Issue #374 - Allow for specification of a file encoding for reading/writing Java files on the file system. There is a new property "javaFileEncoding" on the <context> element that can be used to specify a Java file encoding. (XML files are always read/written in UTF-8 per the spec).
- Added the ability to specify custom code formatters for generated Java and XML files. See new properties on the <context> element for more information.
- Added support for varargs to `org.mybatis.generator.api.dom.java.Parameter` class
- Added support for `synchronized` and `native` to `org.mybatis.generator.api.dom.java.Method` class
- Added support for `transient` and `volatile` to `org.mybatis.generator.api.dom.java.Field` class
- Issue #375 - Added a new plugin - ToStringPlugin - that will generate a toString() method in the model classes. Thanks Alexei and Iwao!
- Issue #233 - added GWT capability to the Serializable plugin
- Issue #564 - support subpackages at the table level
- Issue #590 - new plugin for <cache>. Thanks Jason Bennett!

Version 1.3.1

Bugs 修复

- Always specify <selectKey> order for MyBatis3 - position within the <insert>

is not relevant.
- "rootInterface" was ignored for XMLMAPPER clients
- Fixed bug when a lower case class name is specified as a domain object name
- Fixed issue #174 - incorrect format of order by clause in selectByExampleWithBlobs

改进

- Added a new MyBatis3 generator that generates code based solely on annotations with no generated XML. Configuration settings for this new generator are as follows:
    - The `targetRuntime` attribute of <context> element is **MyBatis3**
    - The `type` attribute of <javaClientGenerator> element is **ANNOTATEDMAPPER**
- Added a new MyBatis3 generator that generates code based on a mix of annotations and generated XML. Configuration settings for this new generator are as follows:
    - The `targetRuntime` attribute of <context> element is **MyBatis3**
    - The `type` attribute of <javaClientGenerator> element is **MIXEDMAPPER**
- Add support for JDBC types NCHAR, NCLOB, NVARCHAR to match MyBatis3.
- Added support for "useGeneratedKeys" in MyBatis3. See <generatedKey> for details.
- Added support for immutable objects and constructor based result maps in MyBatis3. See <table> and/or <javaModelGenerator> for details.
- Added support for initialization blocks in the Java DOM
- Issue #214 - Added the ability to supress all comments in generated code. See <commentGenerator> for details.

Version 1.3.0

1. Moved to mybatis.org, renamed MyBatis Generator
2. MyBatis generator will continue to support XML configuration files from Ibator. However, any new features will only be implemented in MyBatis formatted configuration files. Some minimal changes are required to migrate Ibator formatted configuration files to the new DTD for MyBatis Generator.
3. Configuration settings for MyBatis 3 are as follows:
    i. The `targetRuntime` attribute of <context> element must be changed to **MyBatis3**
    ii. The `type` attribute of <javaClientGenerator> element must be changed to **XMLMAPPER**
4. Removed support for `suppressTypeWarnings` property in the `<context>` element. This confusing property has become unmanagable due to confusion between @SuppressWarnings("unchecked")

and @SuppressWarnings("rawtypes") in the different compilers. This was only used in the corner case where code generated with Ibatis2Java2 targetRuntime was to be compiled with a JSE 5.0 compiler.

Version 1.2.2 (Never Released)

Announcements

- The `org.apache.ibatis.ibator.api.CommentGenerator` interface has changed. Classes that implement this interface **must be changed**. With this change, implementing classes have access to many more data elements from which to generate comments. Additionally, this change makes the comment generator interface more consistent with the other public Ibator interfaces. Details of the change follow:
    - Methods which accepted the parameter `FullyQualifiedTable` now accept the parameter `IntrospectedTable` instead. The `FullyQualifiedTable` instance is available through the method `IntrospectedTable.getFullyQualifiedTable()` .
    - Methods which accepted the `String` parameter `columnName` now accept the parameter `IntrospectedColumn` instead. The column name is available through the method `IntrospectedColumn.getActualColumnName()` .

  重要提示: any implementation that subclasses the Ibator supplied class `DefaultCommentGenerator` does not need to change immediately with this release. The old methods have been deprecated and will be removed with the next release of Ibator - so subclasses should be reworked as soon as convenient.
- The SQL Map generator has changed in that it no longer prepends the string "ibatorgenerated_" to every generated XML id. If you were relying on these generated names in other code, you can force Ibator to prepend the string with the property `useLegacyXMLIds` on the <SqlMapGenerator> configuration.
- Ibator is now built with Maven and includes a Maven plugin

Bugs 修复

- NPE when no DAOs are generated.
- IBATIS-579 - Don't allow column names that contain spaces to break across lines in generated XML.
- Fixed NPE and incorrect calculation in generated equals method (from EqualsHashCodePlugin) when certain fields are null - thanks to Benjamin Klatt for finding this bug.
- IBATIS-601 - improper validation of <generatedKey>

- IBATIS-609 - incorrect parsing of Java generic types
- Fixed spelling error LONCVARCHAR to LONGVARCHAR (thanks Allard)
- Fixed IBATIS-731 - change name of primary key variable to avoid conflicts
- Fixed IBATIS-699 - Overwrite unmergeable XML files if enabled
- Fixed issue where insertSelective failed if there is a sequence generating the primary key (issue only with iBATIS3)

改进

- IBATIS-569 - Modified the IbatorRules implementation to make it easier for plugins to provide custom implementations of IbatorRules. See the Javadoc for the new class `org.apache.ibatis.ibator.internal.rules.IbatorRulesDelegate` for more information.
- IBATIS-571 - Added support for automatically delimiting SQL keywords if they are used as column names in tables. See the <ibatorContext> page for more information.
- IBATIS-577 - Define SQL fragments for column lists to improve reusability of generated code. Thanks to Iwao AVE! for the idea and the initial patch.
- Added new `-verbose` command line argument. See the Running Ibator page for more information.
- Added logging statements for use in debug. See the Logging page for more information.
- Added new example plugin to demonstrate adding case insensitive LIKE support to generated Example classes. See the <ibatorPlugin> page for more information.
- Added "delimitAllColumns" attribute to table configurations. This supports databases like PosgreSQL that are case sensitive for identifiers. See the <table> page for more information.
- Added a page explaining how to deal with case sensitivity in PostgreSQL. See the PostgreSQL page for more information.
- IBATIS-586 - Added the ability to specify nested property elements on columnOverrides. See the <columnOverride> page for more information. Thanks to Dan Turkenkopf for the idea and a nice initial patch.
- The IntrospectedColumn class now contains any column remarks returned during database introspection. This may be useful for some CommentGenerators.
- IBATIS-592 - Added attributes to IntrospectedTable that contain the calculated SqlMap namespace, and the calculated runtime table names. These can now be overridden in plugins.
- Fixed addCriterionfor JDBC* methods so that they all do a null check.
- Fixed IbatorRunner so that configuration errors are shown (thanks to Karel Rank)
- IBATIS-605 - Added Informix Dialect for <generatedKey>
- Addedd support for "distinct" on select by example methods
- Added new "or" method to example classes
- Added new "useCompoundPropertyNames" property on <table>

- Enabled a far simpler method for extending the example classes
- EqualsHashCodePlugin now generates far superior methods

Version 1.2.1

Bugs 修复

- Fixed the IbatorObjectFactory so it will find internal classes from the context classloader.
- Fixed IBATIS-565 - ill formed comment in the SqlMapConfigPlugin

改进

- Modified plugin methods for model fields, getters, and setters so that the plugin will know which type of class (Primary Key, Base Record, or Record with BLOBs) is being generated.
- Added methods to IntrospectedTable to get/set attributes. This allows plugin classes to maintain table based state between plugin calls.
- Added `initialized` method to the plugin API. This allows plugins to alter some of the fundamental code generation items (like the name of a generated class, for example).
- Added an example plugin to show usage of the `initialized` method.

Version 1.2.0

Announcements

- With version 1.2, Abator is renamed to Apache iBATIS Ibator. Several changes have been made to the XML configuration as well as the Java API. See the Migrating from Abator page for detailed information about changes needed to existing Abator configuration files.

Bugs 修复

- Fixed the JavaTypeResolver so that columns with unsupported data types may be overridden by configuration.
- Fixed IBATIS-523 - a bug in the pre-release version of the EqualsHashCodePlugin
- Fixed IBATIS-542 - upgrade the build to Ant version 1.7.1

改进

- Ibator now includes a plugin mechanism. This mechanism can be used to add

to or modify the code generated by Ibator. If you have previously extended one of Abator's code generators to change their behavior, we strongly suggest that you move to a plugin. See the <ibatorPlugin> page for detailed information.
- Ibator ships with the following plugins:
  - A plugin that will generate an SQL Map Config File (`org.apache.ibatis.ibator.plugins.SqlMapConfigPlugin`)
  - A plugin that will make generated model classes Serializable (`org.apache.ibatis.ibator.plugins.SerializablePlugin`)
  - A plugin that will add `equals` and `hashCode` methods to generated model classes (`org.apache.ibatis.ibator.plugins.EqualsHashCodePlugin`)
- Added support for "runtimeCatalog" and "runtimeSchema" properties to the <table> configuration element. Thanks to Dan Turkenkopf for the idea and the patch!
- New generated method - insertSelective. This method will allow you to use column defaults on a table definition on insert
- Added the ability to specify a that DAO implementation classes be in a separate package from the DAO interface classes.

Changes from Abator

There are several breaking changes between Ibator and Abator. This list details the changes, and has methods of resolving the differences.

- JSE 5.0 or higher is required for Ibator
- Ibator does not contain the "legacy" code generators from Abator. You must choose "Ibatis2Java2" or "Ibatis2Java5" as a target runtime - and code generated from Ibator is compatible with iBATIS version 2.2.0 or higher only. If you are using an earlier version of iBATIS - upgrade! If you are not able to upgrade, then you must continue to use Abator.
- The classloading strategy in Ibator is changed from Abator. In all cases, we now recommend specifying the classpath external to Ibator and we further recommend that you do not use the `<classPathEntry>` element. You may specify classpath entries if you feel you must, but those entries will only be used when loading JDBC drivers of Java model root classes. If you write a custom extension to Ibator, or a plugin, you must specify that classpath entry external to Ibator.
- The API for extending Ibator is significantly changed from Abator. In most cases, implementations of the old Abator interfaces should be converted to Ibator plugins. See the Extending Ibator for more information.
- The `afterXXXGenerationHook` methods have been removed from all Ibator supplied implementations of the core interfaces. If you were extending an Ibator supplied implementation to make use of these methods, then you *must* migrate your code to an Ibator plugin.
- The build has been significantly modified and now includes an Emma based code coverage report.

- Changes to the XML configuration file are required. See the Migrating from Abator page for detailed information

Version 1.1.0

Announcements

- The next release of Abator will require JRE 5.0 or higher.
- Java2 is now the default generator set. This will cause different code to be generated if you have not specified a generator set previously. To remedy this, set the generator set to "Legacy".

New Generated Methods

Abator will generate these new methods:

countByExample

This method will return an integer representing the number of rows in a table that match the given criteria.

updateByExample

This method will update all rows in a table that match a given criteria. This method is available in the Java2 and Java5 generator sets only. There is also a "selective" version of the method that only updates certain columns of a table (the selective version of this method is probably the more useful version to use in most situations).

Bugs 修复

- Fixed bug for corner case where a criteria class is created, but no criteria are set.
- Fixed a bug that caused the JavaModelGenerator's "trimStrings" property to fail
- Fixed the XML file merger so that internal entities are preserved
- Fixed the XML configuration parser so that external entities are handled properly
- Fixed bug - incorrect datatype mapping for JDBC BIT datatype
- Fixed bug where Abator generated incorrect properties for certain database columns (for example, if the column name is I_NAME)

Miscellaneous Changes

- Added the ability to specify properties to ignore qualifiers and change runtime table names in the generated SQL for a table. Primary use cases for this

support include:
  - Generating objects for a table that has a public synonym or alias
  - Generating objects for a table that exists in many schemas, and the schema will be selected at runtimeSee the <table> reference page for more information, or the Oracle reference page for an example.
- Added support for delimiting SQL identifiers for the use cases where identifiers contain spaces or SQL reserved words. See the <table>, <abatorContext>, and <columnOverride> reference pages for more information.
- Added SYBASE dialect for generated keys. See the <generatedKey> reference page for more information.
- Added DB2_MF (DB2 on Main Frames) dialect for generated keys. See the <generatedKey> reference page for more information.
- Abator will now automatically escape identifiers that contain the $ or # characters as these characters have special meaning in iBATIS configuration files.
- Added a `clear` method to the generated example classes (in the Java2 and Java5 generator sets only). This allows reuse of these classes.
- Added the ability to specify that result maps should use column indexes rather than column names in the result mappings. Primary use cases for this support include:
  - When tables have columns whose name is only differentiated by case (e.g. "first name" and "First Name")
  - When you want to make the selects as fast as possible (there is a slight performance benefit when using column indexes)See the <table> reference page for more information.
- Made the generated Example and Criteria classes extendable. Added some documentation about how to extend these classes. See the Extending the Example Classes reference page for more information.
- Made the legacy DAOs extendable.
- Added the ability to provide a renaming rule for columns. This is for the use case where columns have a common prefix that should be removed before calculating the property name. See the <columnRenamingRule> reference page for more information.
- Added support for persisting a configuration to XML - this to enable a graphical editor in the future.
- Add afterXXXGenerationHook() methods in all generators to enable adding extra Java code or XML elements to any generated object. This will make it easier to create customized generators.
- API change to allow generating with selected contexts rather than the entire config file.
- API change to allow generating with selected tables rather than the entire config file.
- Exposed new support for selecting tables and/or contexts to the command line and the Ant task - this has added an advanced syntax to the command line for Abator. See the Running Abator reference page for more information.
- `rootClass` and `rootInterface` may now be specified for each table. See the <table> reference page for more information.

- If a `rootClass` is specified for any table, Abator will now check in the rootClass to see if a generated property already exists in the root class. If it does, Abator will not generate the property. The <javaModelGenerator> element now accepts a property to specify the classpath of the `rootClass`. See the <javaModelGenerator> reference page for more information. Thanks to Ashok Madhavan for the beginnings of this code.
- Allowed specifying a type (pre or post) for the generated key element. See the <generatedKey> reference page for more information.
- Added a comment generator interface to enable generation of custom comments. See the <commentGenerator> reference page for more information.

Version 1.0.0

Generator Sets

A generator set is a set of code generators (SQL Map Generator, Java Model Generator, DAO Generator, and Java Type Resolver). Abator now ships three different generator sets. The generated code from these three generator sets is slightly different, and the use of the generated objects is slightly different too. The concepts are exactly the same. With the newer generator sets, the "by example" methods have been vastly improved. It is now possible to generate **virtually any** WHERE clause (including IN and BETWEEN predicates). Lastly, the new generator sets generate much more concise code - the DAOs and SQL Maps are of normal size, and there are no extraneous methods. The example class in the new generator sets encapsulates all the function needed to generate dynamic queries.

The three generator sets shipped with Abator are as follows:

Legacy

This generator set generates code that is the same as previous versions of Abator. There are some limitations with this generator set and we strongly recommend that you choose one of the other sets. However, this set remains the default for now. **This generator set will likely be removed in a future version of Abator.**

Java2

This generator set generates code that is compatible with iBATIS versions 2.2.0 and higher. With this generator set the "by example" methods are much more powerful than in the legacy set. It is now possible to generate virtually unlimited SQL WHERE clauses with Abator generated code (including "IN" and "BETWEEN" clauses). **This generator set will likely become the default set in a future version of Abator.**

Java5

This generator set has the same capabilities as the Java2 generator set with the added feature of generating code that is JSE 5.0 compliant using parameterized types and annotations.

重要**:** code generated with the Java2 or Java5 generator sets is not 100% compatible with code generated with the Legacy set - especially in the use of the "by example" methods. Also note that the "by example" methods in these generator sets rely on iBATIS dynamic SQL support that is missing in iBATIS versions prior to version 2.2.0.

A generator set is selected with the `generatorSet` attribute of the `<abatorContext>` element. See the <abatorContext> reference page for more information.

Use of the example classes is different with the different generator sets. See the Example Class Usage page for more information.

Model Types

A model type is used to give you more control over the types of domain objects generated by Abator. Abator now supports three different types of domain models as follows:

conditional

This model is similar to the hierarchical model except that a separate class will not be generated if that separate class would only contain one field. So if a table has only one primary key field, that field will be merged into the base record class. This model type is the default. **Note that this model type may generate classes that are not 100% with classes generated in previous versions of Abator.**

flat

This model generates only one domain class for any table. The class will hold all fields in the table.

hierarchical

This model is the same as the model shipped with the initial versions of Abator. This model will generate a primary key class if the table has a primary key, another class that holds any BLOB columns in the table, and another class that holds the remaining fields. There is an appropriate inheritance relationship between the classes.

Model types can be specified as a default for an entire context, and you may override the default for each table in a context. See the <abatorContext> reference page for more information about setting the context default.. See the <table> reference page for more information about setting a model type for specific tables.

重要: the default value is conditional - this is a non-backward compatible change from previous versions of Abator.

updateByPrimaryKeySelective

This is a new mapped SQL statement, and new DAO method, that will only update columns whose corresponding properties in the parameter class are non-null. This can be used to update certain columns in a record without needing to update the entire record.

重要: any column that is mapped to a primitive type will always be updated.

Miscellaneous Changes

- Added the ability to specify a table alias. This aids in reuse of generated SQL map elements. See the <table> reference page for more information.
- Fixed the XML file merger so that extraneous blank lines at the end of the file are removed.
- Added the ability to specify a type handler for table columns. See the <columnOverride> reference page for more information.
- Added the ability to specify the visibility of the DAO "by example" methods. This allows you to make the methods private for internal use only. See the <javaClientGenerator> reference page for more information.
- Added the ability to override the naming convention for DAO method names. See the <javaClientGenerator> reference page for more information.
- Added the ability to specify wildcards for schema or tableName in a table configuration. This will allow generation of many tables with a simple XML configuration. See the <table> reference page for more information.
- Added the ability to escape wildcard characters in schema or table names if required by the driver. See the <table> reference page for more information.
- Added the ability to suppress JSE 5.0 type warning messages for non-parameterized types if you are using the Legacy or Java2 generator sets in a JSE 5.0 environment. See the <abatorContext> reference page for more information.
- Added the ability to specify an external properties file for passing parameters into an Abator configuration file (like the iBATIS properties file). See the <properties> reference page for more information.
- The Ant task now supports a "verbose" attribute. See the Running Abator page for more information.
- The Ant task now supports a nested property set for passing values into an Abator configuration file. See the Running Abator page for more information.

MyBatis Generator 快速入门指南

若要启动并快速运行 MyBatis Generator（MBG），请按以下步骤进行:

1. 创建并填写适当的配置文件。 至少，您必须指定:

    i. <jdbcConnection> 元素定义如何连接目标数据库
    ii. <javaModelGenerator> 元素来指定生成 Java 模型对象所属的包
    iii. <sqlMapGenerator> 元素来指定生成 SQL 映射文件所属的包和的目标项目
    iv. (可选的) <javaClientGenerator> 元素来指定目标包和目标项目生成的客户端接口和类（如果您不想生成 Java 客户端代码您可以省略 < javaClientGenerator > 元素）

    查看 XML 配置文件参考 一个配置文件的例子.

2. 将文件保存在一些方便的位置(例如 \temp\generatorConfig.xml)

3. 按下面的方式从命令行运行 MBG:

    ```
    java -jar mybatis-generator-core-x.x.x.jar -configfile
    \temp\generatorConfig.xml -overwrite
    ```

    这会告诉 MBG 使用配置文件去运行。 MBG还会覆盖已经存在的同名Java文件。 如果您想保留已经存在的Java文件，您可以忽略 -overwrite 参数。如果存在冲突, MBG 会用一个唯一的名字保存新生成的文件（例如：MyClass.java.1)。

4. MBG运行后, 您将需要创建或修改的标准 MyBatis 或 iBATIS 配置文件来使用新生成的代码。查看 运行 MyBatis Generator 后的任务 页面获取更多信息。

重要:为iBATIS2生成的代码需要在您的 iBATIS 配置中启用了该命名空间。 查看 运行 MyBatis Generator 后的任务 页面获取更多信息。

运行 MyBatis Generator

MyBatis Generator (MBG) 可以通过以下方式运行:

- 从 命令提示符 使用 XML 配置文件
- 作为 Ant 任务 使用 XML 配置文件
- 作为 Maven Plugin
- 从另一个 Java 程序 使用 XML 配置文件

每种方法的详细信息都在链接的页面上。

注意:还有一个可以良好的集成到Eclipse的MBG插件可以提供额外的功能。 Eclipse 启用 Ant 任务，并支持 Java 文件的自动合并。安装Eclipse插件请参考 MyBatis站点页面的信息。

重要: 当运行在像Eclipse这样的IDE环境的外部运行时, MBG解释 `targetProject` 和 `targetPackage` 的所有XML属性如下:

- `targetProject` 被假定为一个已存在的目录结构。 如果目录结构不存在MBG将会失败。有一个例外 - 当MBG通过Maven插件运行时。 请从Maven插件页面参阅 `targetProject` 在Maven中的详细解释。
- `targetPackage` 将会转换为 `targetProject` 适当的子目录结构。 如果有必要，MBG会创建这些子目录。

从命令行运行 MyBatis Generator

MyBatis Generator (MBG) 可以直接从命令行运行。JAR清单包括默认类的名称(`org.mybatis.generator.api.ShellRunner`)或者您可以自己指定它。 `ShellRunner` 类接受以下的几种详细参数:

参数	值
-configfile *file_name* (必须的)	指定配置文件的名称。
-overwrite (可选的)	如果指定了该参数，如果生成的java文件存在已经同名的文件，新生成的文件会覆盖原有的文件。 如果没有指定该参数，如果存在同名的文件，MBG会给新生成的代码文件生成一个唯一的名字(例如： MyClass.java.1, MyClass.java.2 等等)。 重要: 生成器一定会自动合并或覆盖已经生成的**XML**文件。
-verbose (可选的)	如果指定该参数，执行过程会输出到控制台。
-forceJavaLogging (可选的)	如果指定该参数，MBG将会使用JAVA日志记录而不会使用Log4J,即使Log4J在运行时的类路径中。
-contextids *context1,context2,...* (可选的)	如果指定了该参数，逗号隔开的这些context会被执行。这些指定的context必须和配置文件中 <context> 元素的 `id` 属性一致。 只有指定的这些contextid会被激活执行。如果没有指定该参数，所有的context都会被激活执行。
-tables *table1, table2,...* (可选的)	如果指定了该参数，逗号隔开的这个表会被运行， 这些表名必须和 <table> 配置中的表面完全一致。只有指定的这些表会被执行。 如果没有指定该参数，所有的表都会被执行。 按如下方式指定表明: `table` `schema.table` `catalog..table` 等等。

从命令行运行MGB时您必须指定 XML 配置文件。如果文件的名字是"generatorConfig.xml",可以用下面任意的命令执行:

```
   java -jar mybatis-generator-core-x.x.x.jar -configfile genera
torConfig.xml
   java -jar mybatis-generator-core-x.x.x.jar -configfile genera
torConfig.xml -overwrite
   java -cp mybatis-generator-core-x.x.x.jar org.mybatis.generat
or.api.ShellRunner -configfile generatorConfig.xml
   java -cp mybatis-generator-core-x.x.x.jar org.mybatis.generat
or.api.ShellRunner -configfile generatorConfig.xml -overwrite
```

使用Ant运行 MyBatis Generator

MyBatis Generator (MBG) 包含一个简单的 Ant 任务。这个任务必须定义在您的 build.xml 文件中。任务可以几个参数。这里是一个 build.xml 文件的例子:

```
   <project default="genfiles" basedir=".">
     <property name="generated.source.dir" value="${basedir}" />

     <target name="genfiles" description="Generate the files">
       <taskdef name="mbgenerator"
                classname="org.mybatis.generator.ant.GeneratorAn
tTask"
                classpath="mybatis-generator-core-x.x.x.jar" />
       <mbgenerator overwrite="true" configfile="generatorConfig
.xml" verbose="false" >
         <propertyset>
           <propertyref name="generated.source.dir"/>
         </propertyset>
       </mbgenerator>
     </target>
   </project>
```

MyBatis Generator 任务属性如下:

属性	值
configfile (必须的)	指定配置文件的名称。
overwrite (可选的)	如果指定了该参数，如果生成的java文件存在已经同名的文件，新生成的文件会覆盖原有的文件。 如果没有指定该参数，如果存在同名的文件，MBG会给新生成的代码文件生成一个唯一的名字(例如： MyClass.java.1, MyClass.java.2 等等)。 重要: 生成器一定会自动合并或覆盖已经生成的**XML**文件。
contextids (可选的)	如果指定了该参数，逗号隔开的这些context会被执行。 这些指定的context必须和配置文件中 <context> 元素的 `id` 属性一致。只有指定的这些contextid会被激活执行。如果没有指定该参数，所有的context都会被激活执行。
tables (可选的)	如果指定了该参数，逗号隔开的这个表会被运行， 这些表名必须和 <table> 配置中的表面完全一致。只有指定的这些表会被执行。 如果没有指定该参数，所有的表都会被执行。 按如下方式指定表明: `table` `schema.table` `catalog..table` 等等。
verbose (可选的)	如果设置为 "true", "yes"等等,MBG会将操作信息输出到 ant 控制台 (如果 Ant 在verbose模式下运行)。默认值是 "false"。

注意:

- <taskdef>标记中的类路径用于告诉 Ant MBG JAR 文件在哪里。 这是可选的，除非您将 MBG 添加到 Ant 类路径中 Ant 手册所描述的其他地方。
- 任务的名称可以使任何您想要的, "mbgenerator" 只是一个简单的例子。
- 该任务支持可选嵌套的 < propertyset > 元素，这是标准的 Ant 属性设置的类型。 这可以用于将参数传递到配置文件。例如，上面提到 `generated.source.dir` 这个属性可以在配置文件中使用 `${generated.source.dir}` 进行使用。
- 如果没有指定配置文件中的一个属性，这个属性将会原样输出。

通过Maven运行 MyBatis Generator

MyBatis Generator (MBG) 包含了一个可以集成到Maven构建的Maven插件，按照Maven的配置惯例, 将MBG集成到Maven很容易. 最简配置如下:

```
   <project ...>
     ...
     <build>
       ...
       <plugins>
        ...
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifact
Id>
          <version>1.3.0</version>
        </plugin>
        ...
      </plugins>
      ...
    </build>
    ...
  </project>
```

当然，事情永远不会那么容易！

Maven Goal and Execution（Maven 目标和执行）

The MBG Maven plugin 包含一个目标:

- `mybatis-generator:generate`

这个目标不会被Maven自动执行，他可以通过以下两种方式执行。

可以在命令行通过以下命令执行:

- `mvn mybatis-generator:generate`

您可以通过标准的Maven命令属性传递参数， 例如:

- `mvn -Dmybatis.generator.overwrite=true mybatis-generator:generat`

这条命令会使MBG覆盖重名的文件

在一个连续的编译环境中，您可能想让MGB作为Maven构建的一部分自动执行。这可以通过配置自动执行的目标来实现，这种情况的例子如下:

```
   <project ...>
     ...
     <build>
       ...
       <plugins>
        ...
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifact
Id>
          <version>1.3.0</version>
          <executions>
            <execution>
              <id>Generate MyBatis Artifacts</id>
              <goals>
                <goal>generate</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
        ...
      </plugins>
      ...
    </build>
    ...
  </project>
```

MBG插件将会绑定到Maven构建的 `generate-sources` 阶段。 所以他会在编译步骤之前执行。 此外注意MBG目标将绑定生成Java和XML资源文件的建立，他们都将包括在生成的JAR包内。

MyBatis Generator 配置属性

所有配置在POM中的属性都可以传递到配置文件，并且可以用通常的方式使用。例如:

```
  <project ...>
    ...
    <properties>
      <dao.target.dir>src/main/java</dao.target.dir>
    </properties>
    ...
    <build>
      ...
      <plugins>
       ...
       <plugin>
           <groupId>org.mybatis.generator</groupId>
           <artifactId>mybatis-generator-maven-plugin</artifact
Id>
         <version>1.3.0</version>
         <executions>
           <execution>
             <id>Generate MyBatis Artifacts</id>
             <goals>
               <goal>generate</goal>
             </goals>
           </execution>
         </executions>
       </plugin>
       ...
     </plugins>
     ...
   </build>
   ...
 </project>
```

在这种情况下，属性可以在配置文件中被访问的语法是 `${dao.target.dir}` .

参考参数

所有的参数都是可选的，大部分都适合的默认值。

参数	表达式	类型
configurationFile	${mybatis.generator.configurationFile}	java.io.File
contexts	${mybatis.generator.contexts}	java.lang.String
jdbcDriver	${mybatis.generator.jdbcDriver}	java.lang.String

jdbcDriver	${mybatis.generator.jdbcDriver}	java.lang.String	
jdbcPassword	${mybatis.generator.jdbcPassword}	java.lang.String	
jdbcURL	${mybatis.generator.jdbcURL}	java.lang.String	
jdbcUserId	${mybatis.generator.jdbcUserId}	java.lang.String	
outputDirectory	${mybatis.generator.outputDirectory}	java.io.File	
overwrite	${mybatis.generator.overwrite}	boolean	
sqlScript	${mybatis.generator.sqlScript}	java.lang.String	
tableNames	${mybatis.generator.tableNames}	java.lang.String	
verbose	${mybatis.generator.verbose}	boolean	

targetProject 解释

与Maven运行时 生成器 配置的 `targetProject` 属性有不同的解释。 如果指定值为"MAVEN" (大小写敏感), `targetProject` 将被设置为插件的输出目录，而且如果不存在这个目录，将会创建这个目录。 如果没有设置为 "MAVEN", 那么 `targetProject` 将会被 MGB 当成普通的 - 它必须是一个已经存在的目录。

使用Java运行 MyBatis Generator

MyBatis Generator (MBG) 可以直接使用Java调用。 对于配置，您可以使用XML配置文件，或者完全使用Java进行配置。

使用XML配置文件从Java运行MBG

下面的代码例子展示了如何通过XML配置文件从Java运行MBG。 他不显示异常处理，但是编译错误是很明显的 :)

```
   List<String> warnings = new ArrayList<String>();
   boolean overwrite = true;
   File configFile = new File("generatorConfig.xml");
   ConfigurationParser cp = new ConfigurationParser(warnings);
   Configuration config = cp.parseConfiguration(configFile);
   DefaultShellCallback callback = new DefaultShellCallback(over
write);
   MyBatisGenerator myBatisGenerator = new MyBatisGenerator(conf
ig, callback, warnings);
   myBatisGenerator.generate(null);
```

注意:

- 配置文件属性可以通过 ConfigurationParser 的构造函数的参数传递给解析器。如果没有显式传递，配置文件的属性将会从JVM的系统属性搜索。 例如，属性 `generated.source.dir` 可以在配置文件中通过 `${generated.source.dir}` 被访问。
- 如果没有指定配置文件中的一个属性，这个属性将会原样输出。

通过基于Java的配置运行MGB

下面的代码例子展示了如何通过基于Java的配置运行MGB。 他不显示异常处理，但是编译错误是很明显的 :)

```
   List<String> warnings = new ArrayList<String>();
   boolean overwrite = true;
   Configuration config = new Configuration();

   //   ... fill out the config object as appropriate...

   DefaultShellCallback callback = new DefaultShellCallback(over
write);
   MyBatisGenerator myBatisGenerator = new MyBatisGenerator(conf
ig, callback, warnings);
   myBatisGenerator.generate(null);
```

运行 **MyBatis Generator** 后的任务

在您运行 MyBatis Generator (MBG)后, 您需要创建或者修改其他的 MyBatis 或 iBATIS 配置文件。 主要的任务如下:

- 对 MyBatis 3.x:
  - 创建或修改 MapperConfig.xml 文件
- 对 iBATIS 2.x:
  - 创建或修改 SqlMapConfig.xml 文件
  - 创建或修改 dao.xml 文件 (只有当您使用iBATIS DAO 框架时)

下面将详细地介绍每个任务。

更新 **MapperConfig.xml** 文件 **(MyBatis 3.x)**

MyBatis 3.x 使用了一个 XML 文件, 通常的名字是 `MapperConfig.xml` , 声明数据库连接信息, 事务管理方案, 和 将会用在MyBatis会话中的 XML mapper 文件。 MBG 不会为您创建这个文件，因为它对您执行环境一无所知。然而，一些在此文件中的项目直接涉及 MBG 生成项目。 请参阅标准 MyBatis 数据映射开发指南有关不同的配置选项的详细信息。

MBG 具体需要在配置文件中的有如下:

- 必须列出MBG 生成的 XML 映射文件

例如,假设MBG已经生成一个名为 `MyTableMapper.xml` 的XML映射文件, 并且该文件已经在您的项目的 `test.xml` 包中。 `MapperConfig.xml` 文件应该有这些条目:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>
  <!-- 安装适合于您的环境的事务管理器和数据源 -->
  <environments default="...">
    <environment id="...">
      <transactionManager type="...">
      </transactionManager>
      <dataSource type="...">
      </dataSource>
    </environment>
  </environments>

  <mappers>
    <!-- XML mapper 文件应该在这里列出来 -->
    <mapper resource="test/xml/MyTable_SqlMap.xml" />
  </mappers>

</configuration>
```

如果有多个XML mapper文件(这很常见)，然后您可以通过反复的使用 `<mapper>` 按任意顺序在 `<mappers>` 元素中列出这些文件。

生成 **MapperConfig.xml** 您可能会要MBG使用MapperConfigPlugin生成一个Mapper配置文件的骨架。 查看<插件>页面获取详细信息。

Updating the SqlMapConfig.xml File (iBATIS 2.x)

iBATIS 2 uses an XML file, commonly named `SqlMapConfig.xml` , to specify information for a database connection, a transaction management scheme, and SQL map XML files that will be used in an iBATIS session. MBG cannot create this file for you because MBG knows nothing about your execution environment. However, some of the items in this file relate directly to MBG generated items. Please refer to the standard iBATIS data mapper developer guide for details about the different configuration options.

MBG specific needs in the configuration file are as follows:

- Statement namespaces must be enabled
- MBG generated SQL Map XML files must be listed

For example, suppose that MBG has generated an SQL Map XML file called `MyTable_SqlMap.xml` , and that the file has been placed in the `test.xml` package of your project. The `SqlMapConfig.xml` file should have these entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig
  PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
  "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
  <!-- Statement namespaces are required for MBG -->
  <settings useStatementNamespaces="true" />

  <!-- Setup the transaction manager and data source that are
       appropriate for your environment
  -->
  <transactionManager type="...">
    <dataSource type="...">
    </dataSource>
  </transactionManager>

  <!-- SQL Map XML files should be listed here -->
  <sqlMap resource="test/xml/MyTable_SqlMap.xml" />

</sqlMapConfig>
```

If there is more than one SQL Map XML file (as is quite common), then the files can be listed in any order with repeated `<sqlMap>` elements after the `<transactionManager>` element.

Version 1.2 New Enhancement With MBG version 1.2 and later, You may ask MBG to generate a skeleton SQL Map Configuration file with the SqlMapConfigPlugin. See the <plugin> page for more information.

Updating the dao.xml File (iBATIS 2.x)

重要提示: this step is only required if you generated DAOs for the deprecated iBATIS DAO framework (we suggest using Spring instead).

The iBATIS DAO framework is configured by an xml file commonly called `dao.xml`. The iBATIS DAO framework uses this file to control the database connection information for DAOs, and also to list the DAO implementation classes and DAO interfaces. In this file you should specify the path to your `SqlMapConfig.xml` file, and all the MBG generated DAO interfaces and implementation classes.

For example, suppose that MBG has generated a DAO interface called `MyTableDAO` and a implementation class called `MyTableDAOImpl`, and that the files have been placed in the `test.dao` package of your project. The `dao.xml` file should have these entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE daoConfig
 PUBLIC "-//ibatis.apache.org//DTD DAO Configuration 2.0//EN"
 "http://ibatis.apache.org/dtd/dao-2.dtd">

<daoConfig>
  <context>
    <transactionManager type="SQLMAP">
      <property name="SqlMapConfigResource"
                value="test/SqlMapConfig.xml"/>
    </transactionManager>

    <!-- DAO interfaces and implementations should be listed h
ere -->
    <dao interface="test.dao.MyTableDAO"
         implementation="test.dao.MyTableDAOImpl" />

  </context>
</daoConfig>
```

Migrating from Ibator

This page details changes between MyBatis Generator (MBG) and Ibator. For most users, the changes should be simple. If you extended any of Ibator's classes to supply custom implementations of code generators or the Java type resolver, you will need to rework those custom classes.

Changes are described assuming you are using XML configuration for MBG. If you are using Java based configuration, then the changes are still required and should be easy to deduce from the description of the XML changes.

重要提示: MBG will correctly parse older Ibator configuration files, so no change is actually required. However, new features will only be implemented in configuration files that conform to the MyBatis Generator DTD.

Required for Some Users

- The DTD has changed. The new DOCTYPE should be

  ```
  &lt;!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd"&gt;
  ```

- MyBatis3 is now the default target runtime.
- The `<ibatorConfiguration>` element is renamed to `<generatorConfiguration>`
- The `<ibatorContext>` element is renamed to `<context>`
- The `<ibatorPlugin>` element is renamed to `<plugin>`
- The `<daoGenerator>` element is renamed to `<javaClientGenerator>`
- If you implemented a plugin, the `daoXXX` methods have been renamed to `clientXXX` .

Migrating from Abator

This page details changes between MyBatis Generator (MBG) and Abator. For most users, the changes should be simple. If you extended any of Abator's classes to supply custom implementations of code generators or the Java type resolver, you will need to rework those custom classes.

The changes are listed in three categories: from required configuration changes to less common changes. Note that most changes are described assuming you are using XML configuration for MBG. If you are using Java based configuration, then the changes are still required and should be easy to deduce from the description of the XML changes.

Required for All Users

- The DTD has changed. The new DOCTYPE should be

  ```
  &lt;!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuratio
  n 1.0//EN"
    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd"&
  gt;
  ```

- The `<abatorConfiguration>` element is renamed to `<generatorConfiguration>`
- The `<abatorContext>` element is renamed to `<context>`
- The `<daoGenerator>` element is renamed to `<javaClientGenerator>`

Required for Many Users

- MyBatis3 is now the default target runtime.
- `<context>` element now requires an ID
- The `generatorSet` attribute is removed from the `<context>` element and replaced with the `targetRuntime` attribute. Valid values for this attribute are `Ibatis2Java2`, `Ibatis2Java5`, or `MyBatis3`. MBG does not include the legacy generator set from Abator - so iBATIS version 2.2.0 or higher is required for the code generated by MyBatis Generator.
- The MBG classloading strategy has changed substantially, and we now recommend that you manage the runtime classpath external to MBG. If you manage the classpath with configuration entries, you must make the following changes from Abator:

- Class path entries are specified at the configuration file level with the <classPathEntry> element - now a child element of <generatorConfiguration> only.
- A <classPathEntry> element is not longer allowed as a child of <jdbcConnection>
- The "rootClasspath" property is no longer valid for the <javaModelGenerator> element.

Rarely Required Changes

- The type attribute is removed from both the <javaModelGenerator> and <sqlMapGenerator> elements. MBG has an entirely different method of supplying custom code generators than Abator. See the Extending MyBatis Generator page for full details.
- The JavaTypeResolver interface has changed and is simplified. If you specified a custom implementation on the <javaTypeResolver> element, you must rework your implementation class.
- The ProgressCallback interface has changed significantly. If you implemented this interface for some other execution environment, you will need to rework your implementation.

MyBatis Generator XML 配置参考

最常见的用例中，代码生成器(MBG)是由一个XML配置文件驱动。配置文件告诉MBG:

- 如何连接到数据库
- 生成什么对象，以及如何生成它们
- 那些表生成对象

下面是一个MBG配置文件的例子。查阅每个元素单独的页面查看更多有关元素的更多信息和属性值。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.
0//EN"
  "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>
  <classPathEntry location="/Program Files/IBM/SQLLIB/java/db2ja
va.zip" />

  <context id="DB2Tables" targetRuntime="MyBatis3">
    <jdbcConnection driverClass="COM.ibm.db2.jdbc.app.DB2Driver"
        connectionURL="jdbc:db2:TEST"
        userId="db2admin"
        password="db2admin">
    </jdbcConnection>

    <javaTypeResolver >
      <property name="forceBigDecimals" value="false" />
    </javaTypeResolver>

    <javaModelGenerator targetPackage="test.model" targetProject
="\MBGTestProject\src">
      <property name="enableSubPackages" value="true" />
      <property name="trimStrings" value="true" />
    </javaModelGenerator>

    <sqlMapGenerator targetPackage="test.xml"  targetProject="\M
BGTestProject\src">
      <property name="enableSubPackages" value="true" />
    </sqlMapGenerator>

    <javaClientGenerator type="XMLMAPPER" targetPackage="test.da
o"  targetProject="\MBGTestProject\src">
      <property name="enableSubPackages" value="true" />
    </javaClientGenerator>

    <table schema="DB2ADMIN" tableName="ALLTYPES" domainObjectNa
me="Customer" >
      <property name="useActualColumnNames" value="true"/>
      <generatedKey column="ID" sqlStatement="DB2" identity="tru
e" />
      <columnOverride column="DATE_FIELD" property="startDate" /
>
      <ignoreColumn column="FRED" />
      <columnOverride column="LONG_VARCHAR_FIELD" jdbcType="VARC
HAR" />
    </table>

  </context>
</generatorConfiguration>
```

有关此文件的重要说明:

- 该文件指定旧式DB2 CLI驱动程序将用于连接到数据库，并指定在何处可以找到驱动程序。
- Java类型解析器不应该强制型对象字段BigDecimal的使用，这意味着整数类型(短、整型、长等)如果可能的话将被取代。 此功能是为了使数据库DECIMAL和NUMERIC列容易处理。
- Java模型生成器应该使用子包。 这意味着在这种情况下生成的模型对象将被放置在名为 `test.model.db2admin` 的包中（因为表在 DB2ADMIN schema 中）。 如果 `enableSubPackages` 属性设置为 `false` ，那么包名将会是 `test.model` 。 Java模型生成器也应该对字符串进行trim操作。 这意味着任何字符串属性的setter方法将调用trim方法 - 如果您的数据库可能会在字符末尾返回空白符，这是非常有用的。
- SQL映射生成器将使用子包。这意味着这种情况下生成的XML文件将被放置在名为 `test.xml.db2admin` 的包中（因为表在 DB2ADMIN schema中）。 如果 `enableSubPackages` 属性设置为 `false` ，那么包名将会是 `test.xml` 。
- DAO生成器将使用子包。这意味着这种情况下生成的DAO类都会被放置在名为 `test.dao.db2admin` 的包中（因为表在 DB2ADMIN schema中）。 如果 `enableSubPackages` 属性设置为 `false` ，那么包名将会是 `test.dao` 。DAO生成器会生成一些引用Mybatis XML配置的mapper接口。
- The file specifies only one table will be introspected, but many more could be specified. Important notes about the specified table include:
  - The generated objects will be based on the name Customer (`CustomerKey` , `Customer` , `CustomerMapper` , etc.) - rather than on the table name.
  - Actual column names will be used as properties. If this property were set to `false` (or not specified), then MBG would attempt to camel case the column names. In either case, the name can be overridden by the `<columnOverride>` element
  - The column has a generated key, it is an identity column, and the database type is DB2. This will cause MBG to generate the proper `<selectKey>` element in the generated `<insert>` statement so that the newly generated key can be returned (using DB2 specific SQL).
  - The column `DATE_FIELD` will be mapped to a property called `startDate` . This will override the default property which would be `DATE_FIELD` in this case, or `dateField` if the `useActualColumnNames` property was set to `false` .
  - The column `FRED` will be ignored. No SQL will list the field, and no Java property will be generated.
  - The column `LONG_VARCHAR_FIELD` will be treated as a `VARCHAR` field, regardless of the actual data type.

<classPathEntry> 元素

<classPathEntry> 元素 用于添加运行类路径位置到类路径中的 MyBatis Generator（MBG）。 <classPathEntry> 元素是<generatorConfiguration> 的子元素. MBG 在这些情况下从这些路径下加载类:

- 当加载 JDBC 驱动内省数据库时
- 当加载根类中的 JavaModelGenerator 检查重写的方法时

这个元素是可选的，而且如果您给MGB安装了类路径，您就不需要这个元素(例如使用 `java` 命令时使用参数 `-cp`)。

重要说明: 加载扩展一个 MBG 的类或实现 MBG 的接口之一的类时，不会使用这些位置。 在这种情况下，您必须以相同的方式将 MBG 添加到运行时的类路径下(例如 使用 `java` 命令时使用参数 `-cp`)。

必选属性

属性	描述
location	要添加到类路径中的 JAR/ZIP 文件的完整路径名称或要添加到类路径中的目录。

可选属性

无

子元素

无

例子

这里指定了DB2 JDBC驱动的路径:

```
<classPathEntry location="/Program Files/IBM/SQLLIB/java/db2java
.zip" />
```

<columnOverride> 元素

MyBatis Generator (MBG) 使用 <columnOverride> 元素从将某些属性默认计算的值更改为内省数据库列得的值。 这个元素是 <table> 元素可选的一个子元素。

必选属性

属性	描述
column	要内省的列名

可选属性

属性	描述
property	要使用的 Java 属性的名称。如果没有指定，MBG会根据列名生成。 例如，如果一个表的一列名为"STRT_DTE"， MBG会根据"useActualColumnNames"属性(查看<table> 的属性获取更多的信息) 生成"STRT_DTE" 或 "strtDte"。
javaType	该列属性的完全限定的 Java 类型。如果需要，这可以覆盖由 `JavaTypeResolver` 计算出的类型。 对某些数据库来说， 这是必要的用来处理“奇怪的”数据库类型（例如 MySql 的 unsigned bigint 类型需要映射为 java.lang.Object)。
jdbcType	该列的 JDBC 类型 (INTEGER, DECIMAL, NUMERIC, VARCHAR 等等)。 如果需要，这可以覆盖由 `JavaTypeResolver` 计算出的类型。 对某些数据库来说，这是必要的用来处理怪异的 JDBC 驱动 (例如 DB2 的 LONGVARCHAR 类型需要为iBATIS 映射为 VARCHAR)。
typeHandler	用户定义的需要用来处理这列的类型处理器。 它必须是一个继承iBATIS的 `TypeHandler` 类或 `TypeHandlerCallback` 接口（该接口很容易继承）的全限定的类名。 如果没有指定或者是空白，iBATIS会用默认的类型处理器来处理类型。 重要: MBG 不会校验这个类型处理器是否存在或者可用。 MGB只是简单的将这个值插入到生成的SQL映射的配置文件中。
delimitedColumnName	指定是否应在生成的 SQL 分隔的列名称。 如果列的名称中包含空格，MGB会自动的分割列名， 所以这个重写只有当列名需要强制为一个合适的名字或者列名是数据库中的保留字时是必要的。当设置为 "true" 是, `column` 属性用于重写必须匹配从数据库返回的列名。分隔符指定在 <context> 元素上。默认值是 *false*。然而*MGB*会自动分割包含空格的列名，所以在这种情况下不需要重写。

子元素

- <property> (0..N) 注：在此处定义的任何属性都会被加入相应的内省列的属性集合中。 MBG目前不响应任何属性。 此元素被设置使得对于每一列的特殊值，可以在该插件被编码， 以产生或修改一些独特的特定列中的事件被提供给插件。

<columnRenamingRule> 元素

MyBatis Generator (MBG) 使用 <columnRenamingRule> 元素处理在内省表 中相应的属性名之前重命名数据库列。 这对那些存在同一前缀的字段想在生成属性名时去除前缀的表非常有用。 例子，假设一个表包含以下的列：

- CUST_BUSINESS_NAME
- CUST_STREET_ADDRESS
- CUST_CITY
- CUST_STATE

生成的所有属性名中如果都包含 CUST的前缀可能会让人不爽。这些前缀可以通过如下方式定义重命名规则:

```
&lt;columnRenamingRule searchString="^CUST_" replaceString="" /&gt;
```

注意，在内部，MBG使用 `java.util.regex.Matcher.replaceAll` 方法实现这个功能。 请参阅有关该方法的文档和在Java中使用正则表达式的例子。

当 <columnOverride> 匹配一列时，这个元素（<columnRenamingRule>）会被忽略。 <columnOverride>优先于重命名的规则。

如果指定该选项，重命名规则会在生成属性名前替换列名。 计算出的属性名可能不同，这取决于 <table> 元素上的 "useActualColumnNames"属性。 下表显示了如果重命名规则用于字段集上时不同的值:

列名	属性 **useActualColumnNames="true"**	**useActual**
CUST_BUSINESS_NAME	BUSINESS_NAME	businessNa
CUST_STREET_ADDRESS	STREET_ADDRESS	streetAddre
CUST_CITY	CITY	city
CUST_STATE	STATE	state

这个元素是<table> 元素的一个可选的子元素。

必选属性

属性	描述
searchString	定义将被替换的子字符串的正则表达式。

可选属性

属性	描述
replaceString	这是一个用来替换搜索字符串列每一个匹配项的字符串。如果没有指定，就会使用空字符串。

子元素

无

<commentGenerator> 元素

<commentGenerator> 元素用来定义注释生成器的属性。 注释生成器用来给由MyBatis Generator (MBG) 生成的多种元素（Java自动，Java方法，XML元素等等）生成注释。 默认的注释生成器将 JavaDoc 注释添加到所有生成的Java元素上，从而使Eclipse插件可以使用合并Java的功能。 此外，注释还会添加到每一个生成的XML元素。 注释的目的还在于告诉用户这个元素是生成的，并且有可能重新生成（也就是说 - 他们不应该被修改）。 这个元素是 <context> 元素的一个可选的子元素。

默认的实现类是 `org.mybatis.generator.internal.DefaultCommentGenerator` 。 如果您只想修改某些行为，默认的实现类专门为扩展进行设计。

必选属性

无

可选属性

属性	描述
type	使用用户指定提供的注释生成器类型。 这个类必须继承 `org.mybatis.generator.api.CommentGenerator` 接口。 而且必须有一个默认的构造函数。 这个属性接收指定特殊的值 DEFAULT，这将会使用默认的实现类（这和不指定该属性的效果一样）。

子元素

- <property> (0..N)

支持的属性

这个表格列出了注释生成器中所有可以通过<property>子元素进行设置的属性：

属性名	属性值
suppressAllComments	这个属性用来指定MBG生成的代码中是否包含任何注释。 这个属性有以下可选值：
	false 这是默认值 当这个属性是false或者没有指定时，所有生成的元素都会包含用来说明这是生成元素的注释。
	true 当这个属性是true时，不会往生成的元素中添加任何注释。
	警告: 如果您将这个值设为true，那么所有的代码合并都会被禁用。
suppressDate	这个元素用来指定生成的注释中是否包含生成的日期。 这个属性有以下可选值：
	false 这是默认值 当这个属性是false或者没有指定时，所有元素生成注释时都会带着生成时间。
	true 当这个属性是true时，注释中不会添加时间戳。

示例

这个元素指定了我们不希望生成的注释中包含时间戳:

```
<commentGenerator>
  <property name="suppressDate" value="true" />
</commentGenerator>
```

<context> 元素

<context> 元素用于指定生成一组对象的环境。 子元素用于指定要连接到的数据库、 要生成对象的类型和要内省的表。 多个 <context> 元素可以在 <generatorConfiguration> 元素中列出来，这样可以在同一个MyBatis Generator (MBG)从不同的数据库或者使用不同的生成生成器参数生成对象。

必选属性

属性	描述
id	context唯一的标识符。此值将用于某些错误消息。

可选属性

属性	描述
defaultModelType	这个属性用来设置生成对象类型的默认值。 对象类型 MBG如何生成实体类。 对某些对象类型，MBG会给每 表生成一个单独的实体类。 对另外一些对象类型，MI 据表结构生成不同的类。这个属性有以下可选值：
	conditional 这是默认值 这个模型和hierarchical类似， 果那个单独的类将只包含一个字段，将不会生成一个 类。 因此,如果一个表的主键只有一个字段,那么不会 段生成单独的实体类,会将该字段合并到基本实体类中
	flat 该模型为每一张表只生成一个实体类。这个实体类 表中的所有字段。
	hierarchical 如果表有主键,那么该模型会产生一个单独 键实体类,如果表还有BLOB字段， 则会为表生成一个 有BLOB字段的单独的实体类,然后为所有其他的字段 个单独的实体类。 MBG会在所有生成的实体类之间维 继承关系（注：BLOB类 继承 其他字段类 继承 主键
targetRuntime	此属性用于指定生成的代码的运行时目标。 该属性支 特殊的值：
	MyBatis3 这是默认值 使用这值的时候，MBG会生成 MyBatis 3.0或更高版本， 兼容JSE 5.0或更高版本的 （例如Java模型类和Mapper接口会使用泛型）。 这些 对象中的"by example"方法将支持几乎不受限制的动 where子句。 另外，这些生成器生成的Java对象支持 5.0特性，包含泛型和注解。

	MyBatis3Simple 这是默认值 使用这值的时候，和上面 MyBatis3类似，但是不会有"by example"一类的方法 少量的动态SQL。 Ibatis2Java2 使用这值的时候，ME 成兼容iBATIS 2.2.0或更高版本（除了iBATIS 3），还 Java2的所有层次。 这些生成对象中的"by example"方 支持几乎不受限制的动态的where子句。 这些生成的 能100%和原生的Abator或其他的代码生成器兼容。
	Ibatis2Java5 使用这值的时候，MBG会生成兼容iBAT 2.2.0或更高版本（除了iBATIS 3），兼容JSE 5.0或更 本的对象（例如Java模型类和Dao类会使用泛型）。 成对象中的"by example"方法将支持几乎不受限制的动 where子句。 另外，这些生成器生成的Java对象支持 5.0特性，包含泛型和注解。 这些生成的对象不能100 生的Abator或其他的代码生成器兼容。
	如果您想创建一个完全不同的代码生成器， 使用一个 了 `org.mybatis.generator.api.IntrospectedTab` 的权限定类名替换该值。 通过这个值，您可以创建您 代码生成器，然后插入到代码生成器引擎中。 查阅 扩 MyBatis Generator 页面获取更多信息。
introspectedColumnImpl	使用这个值去指定一个继承 了 `org.mybatis.generator.api.IntrospectedCol` 的权限定名称。 这可以修改代码生成器计算列信息时 为。 查阅 扩展 MyBatis Generator 页面获取更多信息

子元素

- <property> (0..N)
- <plugin> (0..N)
- <commentGenerator> (0 or 1)
- <jdbcConnection> (1 Required)
- <javaTypeResolver> (0 or 1)
- <javaModelGenerator> (1 Required)
- <sqlMapGenerator> (0 or 1)
- <javaClientGenerator> (0 or 1)
- <table> (1..N)

支持的属性

下面的表格列出了所有可用的 <property> 子元素:

属性名	属性值
autoDelimitKeywords	如果是true，那么MGB会分隔SQL关键字，如果他们被用的列名。 MBG维护了许多不同数据库的SQL关键字列表而，列表可能不是非常的全面。 如果一个特殊的关键字没MBG的关键字列表中， 您需要通过 `<columnOverride>` 强制分隔列。查看 `org.mybatis.generator.internal.db.SqlReservedW` 这个类的源码查看MGB包含的关键字列表。默认值是*fals*
beginningDelimiter	要用作需要分隔符的 SQL 标识符开头的标识符分隔符的 MBG会自动分割包含空格的SQL标识符。 MGB还会自动 <table> 或 <columnOverride>中配置了具体要求的标识符值是双引号 *(")*。
endingDelimiter	要使用作为结束的标识符分隔符需要分隔符的 SQL 标识 MBG会自动分割包含空格的SQL标识符。 MGB还会分隔<table> 或 <columnOverride>中配置了具体要求的标识符值是双引号 *(")*。
javaFileEncoding	通过这个属性设置要使用的Java文件的编码。 新生成的J会用这个编码写入到文件系统。 如果Java文件已经存在使用这个编码进行读取，就会执行合并操作。 如果没有会使用平台的默认编码。查阅 `java.nio.charset.Char` 取可以选择的编码的信息。
javaFormatter	使用此属性来指定生成的 Java 文件的用户提供 formater 的类名称。 这个类必须继承 `org.mybatis.generator.api.JavaFormatter` 而且必个默认（不含参数）的构造方法。 每一个context都持有的javaFormatter的实例。 默认的javaFormatter是 `org.mybatis.generator.api.dom.DefaultJavaForma` 默认的格式使用内置到Java DOM类的格式。
xmlFormatter	使用此属性来指定生成的 XML 文件的用户提供 formater 的类名称。 这个类必须继承 `org.mybatis.generator.api.XmlFormatter` 而且必默认（不含参数）的构造方法。 每一个context都持有一xmlFormatter的实例。 默认的xmlFormatter是 `org.mybatis.generator.api.dom.DefaultXmlFormat` 默认的格式使用内置到Java DOM类的格式。

<generatedKey> 元素

<generatedKey> 元素用来指定自动生成主键的属性（identity字段或者sequences序列）。如果指定了这个元素，MyBatis Generator (MBG)会在生成insert的SQL映射文件中插入一个合适的 <selectKey> 元素。这个元素是 <table> 元素的一个可选的子元素。

必选属性

属性	描述
column	生成列的列名。
sqlStatement	将返回新值的 SQL 语句。如果这是一个identity列，您可以使用其定义的的特殊值。或者为您的数据库使用一个合适的语句。预先特殊值如下:
	Cloudscape 这将转化为: `VALUES IDENTITY_VAL_LOCAL()`
	DB2 这将转化为: `VALUES IDENTITY_VAL_LOCAL()`
	DB2_MF 这将转化为:`SELECT IDENTITY_VAL_LOCAL()
	FROM SYSIBM.SYSDUMMY1`为运行在zOS（主框架）或者有些iSeries (AS/400)上的DB2数据库使用该值
	Derby 这将转化为: `VALUES IDENTITY_VAL_LOCAL()`
	HSQLDB 这将转化为: `CALL IDENTITY()`
	Informix 这将转化为: `select dbinfo('sqlca.sqlerrd1') from systables where`
	MySql 这将转化为: `SELECT LAST_INSERT_ID()`
	SqlServer 这将转化为: `SELECT SCOPE_IDENTITY()`
	SYBASE 这将转化为: `SELECT @@IDENTITY`
	JDBC 这会配置MBG使用MyBatis3支持的JDBC标准的生成key来码。这是一个独立于数据库获取标识列中的值的方法。
	重要: 只有当目标运行为MyBatis3时才会产生正确的代码。如果一起使用目标运行时会产生运行时错误的代码。

可选属性

属性	描述
identity	当设置为 `true` 时,该列会被标记为identity列， 并且 `<selectKey>` 元素会被插入在insert后面。 当设置为 `false` 时， `<selectKey>` 会插入到insert之前（通常是序列）。重要: 即使您 `type` 属性指定为"post"， 您仍然需要为identity 列将该参数设置为 "true"。 这将标志MBG从插入列表中删除该列。默认值是 *false*.
type	如果指定，则此值将被添加为生成selectKey元素的类型。 此属性的值应该是"pre"或"post"。重要: 如果指定此属性的值，然后生成selectKey元素将永远被放置在insert语句之前。

子元素

无

<generatorConfiguration> 元素

<generatorConfiguration> 元素是MyBatis Generator配置的根元素。这个文件必须包含下面的 DOCTYPE:

```
<!DOCTYPE generatorConfiguration PUBLIC
  "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
  "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
```

必选属性

无

可选属性

无

子元素

- <properties> (0 or 1)
- <classPathEntry> (0..N)
- <context> (1..N)

<ignoreColumn> 元素

<ignoreColumn>元素用来告诉 MyBatis Generator (MBG) 忽略一个表中的列。 生成的SQL不会使用该列，而且生成的模型对象中将不包含该列。 这个元素是<table> 元素的一个可选的子元素。

必选属性

属性	描述
column	被忽略的列名。

可选属性

属性	描述
delimitedColumnName	当匹配对从数据库中返回的列时，如果为真，则MBG将执行区分大小写完全匹配。 如果为false(默认值)，则被认为是不区分大小写的名称。

子元素

无

<javaClientGenerator> 元素

<javaClientGenerator> 元素用于定义 Java 客户端代码生成器的属性。 Java客户端生成器生成Java接口和类，它可以简单地使用生成的Java模型和XML映射文件。在iBATIS2目标环境中，这些对象可以生成DAO接口和实现类形式的代码。 对Mybatis来说，会生成Mapper接口形式的代码。 这个元素是 <context> 元素的一个可选的子元素。 如果不指定此元素，然后代码生成器(MBG)将不会生成Java客户端接口和类

必选属性

属性	描述
type	此属性用于选择一个预定义的Java客户端的生成器，或指定用户Java客户端生成器。 任何用户提供的DAO生成器必须继承 `org.mybatis.generator.codegen.AbstractJavaClientGen` 类, 必须有一个公开默认的构造函数。该属性接收下列预定义的生成器之一:
	如果<context> targetRuntime 是 **MyBatis3**:
	ANNOTATEDMAPPER 生成的对象是 MyBatis 3.x 映射器基础结 Java接口。 这些接口将会基于注解和 MyBatis 3.x SqlProviders 有XML文件生成。（注：就是纯接口使用注解的形式，不会有X件）ANNOTATEDMAPPER 依赖 MyBatis 3.0.4 或更高版本。
	MIXEDMAPPER 生成的对象是 MyBatis 3.x 映射器基础结构的口。 这些接口将基于注解和XML的混合形式。 注解将会用在简可以实现的地方。 此客户端不会生成SqlProvider，所有复杂的都会生成在XML中。The MIXEDMAPPER 依赖 MyBatis 3.0.4 本。
	XMLMAPPER 生成的对象是 MyBatis 3.x 映射器基础结构的 Ja口。 这些接口将会依赖于生成的XML文件。
	如果 <context> targetRuntime 是 **MyBatis3Simple**:
	ANNOTATEDMAPPER 生成的对象是 MyBatis 3.x 映射器基础结 Java接口。 这些接口将会基于注解和 MyBatis 3.x SqlProviders 有XML文件生成。（注：就是纯接口使用注解的形式，不会有X件）The ANNOTATEDMAPPER 依赖 MyBatis 3.0.4 或更高版本
	XMLMAPPER 生成的对象是 MyBatis 3.x 映射器基础结构的 Ja口。 这些接口将会依赖于生成的XML文件。
	如果 <context> targetRuntime 是 **Ibatis2Java2** or **Ibatis2Java**
	IBATIS 生成的对象将符合（不建议使用）iBATIS的DAO框架。

	GENERIC-CI 生成的对象将只依赖于SqlMapClient。SqlMapCli 通过构造参数依赖注入提供。 生成的对象将是DAO接口和实现 式。
	GENERIC-SI 生成的对象将只依赖于SqlMapClient。SqlMapCli 通过setter方法依赖注入提供。 生成的对象将是DAO接口和实现 式。
	SPRING 生成的对象将符合Spring的DAO框架。
targetPackage	这个包用于放置生成的接口和实现类。 在默认的生成器中，属性"enableSubPackages"用来控制如何计算实际的包。 如果是tr 果表的catalog和schema存在，就将他们作为子包加起来。 如果"enableSubPackages"是false（默认值），计算的package将 targetPackage属性指定的值。 MBG 将创建所需的生成包的文 注：实现类的包可以通过指定下面会提到的可选的 `implementationPackage` 属性。
targetProject	这用来指定生成接口和类的目标项目。 当在Eclipse环境中运行 项指定保存对象的位置的项目和源文件夹。 在其他环境中，此 地文件系统上的现有目录。 如果它不存在，MBG不会创建它。

可选属性

属性	描述
implementationPackage	如果指定了该属性，实现类就会生成在这个包中。 在默认的生成器中，属性"enableSubPackages"用来控制如何计算实际的包。 如果是true，如果表的catalog和schema存在，就将他们作为子包加起来。 如果"enableSubPackages"是false（默认值），计算的package将是targetPackage属性指定的值。 MBG 将创建所需的生成包的文件夹。

子元素

- <property> (0..N)

支持的属性

这个表格列出了所有可以通过<property>子元素进行设置的属性：

属性名	属性值
	这个属性用来选择MGB是否根据基于目录和内省表来 Java包。例如，假设某个表MYTABLE在MYSCHMA的

enableSubPackages	中，也假设targetPackage属性的值设置为"com.myco 果此属性为true，为这个表生成的DAO接口和类将被方 包"com.mycompany.myschema"中。 如果此属性是fa SQL映射将被放在"com.mycompany" schema中。默认
exampleMethodVisibility	此属性用来设置不同"ByExample"方法的可见性 - selectByExample, deleteByExample等等。 如果没有 方法将会是公开的的，并将在接口中声明。 此属性使 这些方法如果您只想使用它们来执行其他专门的方法
	public 这是默认值 生成实现类中的方法将是公开的， 声明的方法。
	private 生成实现类中的方法将是private的，也不会在 明。
	protected 生成实现类中的方法将是protected的，也不 声明。
	default 生成实现类中的方法将是包内可见的，也不会 明。
	重要提示: 如果目标运行时是MyBatis3，则将忽略此属
methodNameCalculator	此属性用于选择一个方法名称计算器。 方法名称计算 为DAO方法提供不同的名称。 您可以选择一个预定义 提供的选项都适合您的环境， 也可以指定一个实现 了 `org.mybatis.generator.api.DAOMethodNameCa` 口的 完全限定名称。
	default 这是默认值 生成的方法名称会很简单("insert", "updateByPrimaryKey"等等)。
	extended 生成的方法名称将包括与方法关联的实体对 ("insertWidget", "updateWidgetByPrimaryKey"等等)。
	重要提示: this property is ignored if the target runtime MyBatis3.
rootInterface	此属性用于指定一个所有生成的接口都继承的父接口 以通过表配置的 `rootInterface` 属性覆盖。重要: M 接口是否存在或者是否是一个有效的接口。如果指定 应该是一个全限定接口名称(例如 com.github.abel533.mapper.Mapper)。
useLegacyBuilder	如果为true，那么客户端将使用SqlBuilder为MyBatis生 SQL。 在MyBatis 3.2或以后版本，这个SqlBuilder废弃 个新的SQL类替代。 如果是false，MBG会使用新的S 成客户端代码。默认值是 *false*

示例

此元素指定我们总希望生成的接口和对象在"text.model'包中，并且我们希望使用基于schema和catalog的子包。 它还指定了我们想使用为MyBatis3的XML配置文件的mapper接口。

```
<javaClientGenerator targetPackage="test.model"
     targetProject="\MyProject\src" type="XMLMAPPER">
  <property name="enableSubPackages" value="true" />
</javaClientGenerator>
```

The <javaModelGenerator> Element

The <javaModelGenerator> element is used to define properties of the Java model generator. The Java Model Generator builds primary key classes, record classes, and Query by Example classes that match the introspected table. This element is a required child element of the <context> element.

必选属性

属性	描述
targetPackage	This is the package where the generated classes will be placed. In the default generator, the property "enableSubPackages" controls how the actual package is calculated. If true, then the calculated package will be the targetPackage plus sub packages for the table's catalog and schema if they exist. If false (the default) then the calculated package will be exactly what is specified in the targetPackage attribute. MyBatis Generator will create folders as required for the generated packages.
targetProject	This is used to specify a target project for the generated objects. When running in the Eclipse environment, this specifies the project and source folder where the objects will be saved. In other environments, this value should be an existing directory on the local file system. MyBatis Generator will not create this directory if it does not exist.

可选属性

无

子元素

- <property> (0..N)

支持的属性

下面的表格列出了所有可用的 <property> 子元素:

属性名	属性值

constructorBased	This property is used to select whether MyBatis Generator will generate a constructor for the class that accepts a value for each field in the class. Also, the SQL result map will be built to use the constructor rather than the "setter" for each field.This property is only applicable for MyBatis3 and will be ignored for iBATIS2.This property is ignored (and forced "true") if the "immutable" property is set "true".This property can be overridden by the corresponding property in a <table> element.默认值是 *false*
enableSubPackages	This property is used to select whether MyBatis Generator will generate different Java packages for the objects based on the catalog and schema of the introspected table.For example, suppose a table MYTABLE in schema MYSCHMA. Also suppose that the targetPackage attribute is set to "com.mycompany". If this property is true, the generated objects for the table will be placed in the package "com.mycompany.myschema". If the property is false, the generated objects will be placed in the "com.mycompany" schema.默认值是 *false*
immutable	This property is used to select whether MyBatis Generator will generate immutable model classes - this means that the classes will not have "setter" methods and the constructor will accept values for each field in the class.If true, this forces the model classes to be built with paramterized constructors regardless of the value of the "constructorBased" property.This property is only applicable for MyBatis3 and will be ignored for iBATIS2.This property can be overridden by the corresponding property in a <table> element.默认值是 *false*
rootClass	This property can be used to specify a root class for all generated Java model objects. MyBatis Generator will specify this value as the super class of the primary key object, if the table has a primary key, or the record object otherwise. This value may be overridden by specifying the `rootClass` property on a Table configuration.重要**:** If MyBatis Generator is able to load the root class, then it will not override a property in the root class that exactly matches a property that would normally be generated. An exact match of a property is defined as follows
	Property name matches exactly
	Property is of the same type

	Property has a "getter" method
	Property has a "setter" method
	If specified, the value of this property should be a fully qualified class name (like com.mycompany.MyRootClass).
trimStrings	This property is used to select whether MyBatis Generator adds code to trim the white space from character fields returned from the database. This can be useful if your database stores data in CHAR fields rather than VARCHAR fields. When true, MyBatis Generator will insert code to trim character fields.默认值是 *false*

示例

This element specifies that we always want to place generated classes in the "'test.model" package and that we want to use subpackages based on the table schema and catalog. We also want String columns trimmed.

```
<javaModelGenerator targetPackage="test.model"
     targetProject="\MyProject\src">
  <property name="enableSubPackages" value="true" />
  <property name="trimStrings" value="true" />
</javaModelGenerator>
```

The <javaTypeResolver> Element

The <javaTypeResolver> element is used to define properties of the Java Type Resolver. The Java Type Resolver is used to calculate Java types from database column information. The default Java Type Resolver attempts to make JDBC DECIMAL and NUMERIC types easier to use by substituting Integral types if possible (Long, Integer, Short, etc.) If this behavior is undesirable, set the property "forceBigDecimals" to "true". You can also substitute your own implementation if you want different behavior than the default. This element is an optional child element of the <context> element.

必选属性

无

可选属性

属性	描述
type	This can be used to specify a user provided Java Type Resolver. The class must implement the interface `org.mybatis.generator.api.JavaTypeResolver`，必须有一个公开默认的构造函数。. The attribute also accepts the special value DEFAULT in which case the default implementation will be used (this has the same effect as not specifying the type).

子元素

- <property> (0..N)

支持的属性

下面的表格列出了所有可用的 <property> 子元素:

属性名	属性值
forceBigDecimals	This property is used to specify whether MyBatis Generator should force the use of `java.math.BigDecimal` for DECIMAL and NUMERIC fields, rather than substituting integral types when possible. 这个属性有以下可选值：
	false 这是默认值 当这个属性是false或者没有指定时， the default Java type resolver will attempt to make JDBC DECIMAL and NUMERIC types easier to work with by substituting Integral types if possible. The substitution rules are as follows:
	If the scale is greater then zero, or the length is greater than 18, then the java.math.BigDecimal type will be used
	If the scale is zero, and the length is 10 through 18, then the Java type resolver will substitute a java.lang.Long.
	If the scale is zero, and the length is 5 through 9, then the Java type resolver will substitute a java.lang.Integer.
	If the scale is zero, and the length is less than 5, then the Java type resolver will substitute a java.lang.Short.
	true 当这个属性是true时， the Java type resolver will always use java.math.BigDecimal if the database column is of type DECIMAL or NUMERIC.

示例

This element specifies that we always want to use the java.math.BigDecimal type for DECIMAL and NUMERIC columns:

```
<javaTypeResolver>
  <property name="forceBigDecimals" value="true" />
</javaTypeResolver>
```

<jdbcConnection> 元素

<jdbcConnection> 元素用于指定数据库连接所需的内省表的属性。 MyBatis Generator 使用JDBC的数据库元数据类来发现您在配置中指定的属性表。 每一个<context>都必须有一个<jdbcConnection>元素。

必选属性

属性	描述
driverClass	用于访问数据库的JDBC驱动程序的完全限定类名称。
connectionURL	用于访问数据库的JDBC连接URL。

可选属性

属性	描述
userId	访问数据库的用户ID。
password	访问数据库的密码。

子元素

- <property> (0..N) 注意: 这里指定的任何属性将添加到JDBC驱动程序的属性中。

示例

此元素将连接到DB2数据库配置为MBGTEST在DB2客户端配置实用程序。 使用JDBC驱动程序的默认安装位置:

```
<jdbcConnection driverClass="COM.ibm.db2.jdbc.app.DB2Driver"
    connectionURL="jdbc:db2:MBGTEST"
    userId="db2admin"
    password="db2admin">
</jdbcConnection>
```

<plugin> 元素

<plugin> 元素用来定义一个插件。插件用于扩展或修改通过MyBatis Generator (MBG)代码生成器生成的代码。 这是 <context> 元素的一个子元素。 可以在context指定任意数量的插件。插件将在配置中配置的顺序执行。

有关执行插件的详细信息，请参阅 Implementing Plugins 页面的引用。

有关MyBatis Generator提供的插件代码生成器的详细信息，请参阅 Supplied Plugins提供参考页。

必选属性

属性	描述
type	实现该接口的类的完全限定名的插件。该类必须实现该接口 `org.mybatis.generator.api.Plugin` ,必须有一个公开默认的构造函数。注意，继承 `org.mybatis.generator.api.PluginAdapter` 这个适配器类比继承接口更容易扩展。

可选属性

无

子元素

- <property> (0 or 1)

<properties> 元素

<properties> 元素用于指定一个需要在配置中解析使用的外部属性文件。 属性配置中的任何属性将接受 `${property}` 这种形式的属性。 从指定的属性文件中搜索匹配的值，配置的值将会被替换。 属性文件时正常的Java属性文件的格式。

<properties>元素是 <generatorConfiguration> 元素的子元素。

必选属性

以下属性是必需的，并且只能有其中一个。

属性	描述
resource	属性文件的全限定名称。当指定了resource属性，将会从classpath下面搜索属性文件。 当指定为 `com/myproject/generatorConfig.properties` 时，他必须存在 `com.myproject` 包下面。
url	属性文件的URL值。这可以用于指定一个属性文件在文件系统上的特定位置，例如 `file:///C:/myfolder/generatorConfig.properties` 。

可选属性

无

子元素

无

<property> 元素

<property> 元素用于指定许多其他元素的属性。 每个支持属性元素的文档页面列出的不同值都是有效的。 此元素也可以用于将属性传递给您的任何自定义的代码生成器执行。

必选属性

属性	描述
name	属性的名称(区分大小写)。
value	属性的值(通常是不区分大小写)。

可选属性

无

子元素

无

<sqlMapGenerator> 元素

<sqlMapGenerator> element 用于定义SQL映射生成器的属性。 SQL映射生成器为每一个内省的表生成MyBatis/iBATIS形式的XML文件。

如果目标是iBATIS2，那么这个元素是<context> 元素的一个必须的子元素。

如果目标是MyBatis3，那么只有当您选择javaClientGenerator需要XML时，他才是<context> 元素的一个必须的子元素。

如果不指定一个javaClientGenerator，则适用以下规则:

- 如果您指定一个sqlMapGenerator，那么MBG将只生成SQL映射的XML文件和模型对象。
- 如果您没有指定一个sqlMapGenerator，那么MBG将只会生成模型对象。

必选属性

属性	描述
targetPackage	这个包用于放置生成的SQL映射文件。 在默认的生成器中，属性"enableSubPackages"用来控制如何计算实际的包。 如果是true，如果表的catalog和schema存在，就将他们作为子包加起来。 如果"enableSubPackages"是false（默认值），计算的package将是targetPackage属性指定的值。 MBG 将创建所需的生成包的文件夹。
targetProject	这用来指定生成SQL映射的目标项目。 当在Eclipse环境中运行时，此选项指定保存对象的位置的项目和源文件夹。 在其他环境中，此值应为本地文件系统上的现有目录。 如果它不存在，MBG不会创建它。

可选属性

无

子元素

- <property> (0..N)

支持的属性

下面的表格列出了所有可用的 <property> 子元素:

属性名	属性值
enableSubPackages	这个属性用来选择MGB是否根据基于目录和内省表来生成不同的Java包。例如，假设某个表MYTABLE在MYSCHMA的schema中，也假设targetPackage属性的值设置为"com.mycompany"。 如果此属性为true，为这个表生成的DAO接口和类将被放置在包"com.mycompany.myschema"中。 如果此属性是false，生成的SQL映射将被放在"com.mycompany" schema中。默认值是 *false*

示例

此元素指定我们总希望生成的SQL映射文件在"text.model'包中，并且我们希望使用基于schema和catalog的子包。 它还指定了我们想使用为MyBatis3的XML配置文件的mapper接口。

```
<sqlMapGenerator targetPackage="test.model"
     targetProject="\MyProject\src">
  <property name="enableSubPackages" value="true" />
</sqlMapGenerator>
```

可选属性

属性	描述
schema	数据库 schema - 如果您的数据库不使用 schema ，或者有一个默认的 schema 您不需要设置 schema。 如果需要，指定的值可以包含SQL通配符。
catalog	数据库 catalog - 如果您的数据库不使用 catalog，或者有一个默认的 catalog，您就不需要设置 catalog。
alias	如果指定，这个值会用在生成的select查询SQL的表的别名和列名上。 列名会被别名为 alias*actualColumnName(*别名实际列名) 这种模式。
domainObjectName	生成对象的基本名称。如果没有指定，MBG会自动根据表名来生成名称。 这个（指定或者自动生成的）名字将用于计算实体类的名称和DAO类的名称。您可以在实体对象的名字上指定包名分割。 例如，您可以指定`foo.Bar`，然后实体对象的名字会是`Bar`，包名`foo`会添加到生成器配置中指定的目标包后面。
enableInsert	指定是否生成Insert语句。默认值是 *true*。
enableSelectByPrimaryKey	指定是否生成通过主键查询的语句。 无论这个怎么设置，当表不存在主键的时候，不会生成这个语句。默认值是 *true*。
enableSelectByExample	指定是否生成通过Example查询的语句。 这个语句支持运行时生成多种不同条件的动态查询。默认值是 *true*。
enableUpdateByPrimaryKey	指定是否生成通过主键更新的语句。 无论这个怎么设置，当表不存在主键的时候，不会生成这个语句。默认值是 *true*。
enableDeleteByPrimaryKey	指定是否生成通过主键删除的语句。 无论这个怎么设置，当表不存在主键的时候，不会生成这个语句。默认值是 *true*。
enableDeleteByExample	指定是否生成通过Example删除的语句。 这个语句支持运行时生成多种不同的条件动态删除。默认值是 *true*。
enableCountByExample	指定是否生成通过Example查询总数的语句。 这个语句将返回满足Example条件的数据总数。默认值是 *true*。
enableUpdateByExample	指定是否生成通过Example更新的语句。 这个语句将更新满足Example条件的数据。 如果设置为True,UpdateByExampleSelective语句也会生成。 这个语句只会更新那些参数中值不为null的的列。默认值是 *true*。
selectByPrimaryKeyQueryId	这个值会以"'<value>' as QUERYID"这种形式被添加到通过主键查询的语句的select列中。 这可能对在运行时的DBA跟踪工具中标记查询有用。 如果您使用这个值，您需要为MBG生成的每一个查询指定一个唯一的id。

selectByExampleQueryId	这个值会以"'<value>' as QUERYID"这种形式被添加到通过Example查询的语句的select列中。 这可能对在运行时的DBA跟踪工具中标记查询有用。 如果您使用这个值，您需要为MBG生成的每一个查询指定一个唯一的id。
modelType	如果您需要，这个值可以用来重写默认的模型类型。 如果没有指定，MBG将会生成基于上下文默认模型类型的实体对象。 模型类型定义了MBG如何生成实体类。 有一些模型类型MGB会为每个表生成一个单独的实体类。 另外一些模型，MGB会根据表结构生成不同的一些类。 这个属性有以下可选值：
	conditional 这个模型和hierarchical类似，除了如果那个单独的类将只包含一个字段，将不会生成一个单独的类。 因此,如果一个表的主键只有一个字段,那么不会为该字段生成单独的实体类,会将该字段合并到基本实体类中。
	flat 该模型为每一张表只生成一个实体类。这个实体类包含表中的所有字段。
	hierarchical 如果表有主键,那么该模型会产生一个单独的主键实体类,如果表还有BLOB字段， 则会为表生成一个包含所有BLOB字段的单独的实体类,然后为所有其他的字段生成一个单独的实体类。 MBG会在所有生成的实体类之间维护一个继承关系（注：BLOB类 继承 其他字段类 继承 主键类）。
escapeWildcards	这个属性表示当查询列，是否对schema和表名中的SQL通配符 (*'' and '%')* 进行转义。 对于某些驱动当*schema*或表名中包含*SQL*通配符时（例如，一个表名是*MY_TABLE*，有一些驱动需要将下划线进行转义）是必须的。默认值是 _*false*.
delimitIdentifiers	这个属性表示当查询表并且在生成的SQL中分隔标识符时，是否使用指定的确切的值。 有关更多详细信息，请参见上面的详述。分隔符在 <context> 元素上指定。默认值是 *false*。除非 `catalog`, `schema` 或 `tableName` 属性值包含空白时，是 *true*.
delimitAllColumns	指示是否给生成SQL中所有的列名添加分隔符。 这是一种给每个列添加`<columnOverride>`来指定列需要被分隔的替代方式。 这对类似PostgreSQL这种使用小写标识符的数据库很有用。分隔符在 <context> 元素上指定。默认值是 *false*.

子元素

- <property> (0..N)
- <generatedKey> (0 or 1)
- <columnRenamingRule> (0 or 1)
- <columnOverride> (0..N)
- <ignoreColumn> (0..N)

元素 元素

元素用来从内省数据库中选择一个表。选定的表会生成如下的这些对象： 一个MyBatis/iBATIS 格式的SQL映射文件 一组根据表生成的"model"类: 一个和表的主键匹配的类(如果表有主键)。 一个包含了非主键字段和BLOB字段的类，如果存在

支持的属性

下面的表格列出了所有可用的 <property> 子元素:

属性名	属性值
constructorBased	此属性用于选择代码生成器是否生成接受类中的每个字段的值的类的构造函数。 此外，SQL结果映射也会为每个字段生成使用构造方法而不是"setter"的方式。此属性仅适用于MyBatis3，iBATIS2将会被忽略。如果"immutable"属性设置为"true"，这个属性将会被忽略。默认值是 *false*
ignoreQualifiersAtRuntime	如果设置为true，MBG不会讲schema或catalog添加到生成SQL中的表名上。 如果您有几个schemas中的表具有相同的名称，这是非常有用的。 您可以使用MBG生成基于在一个schema上的表，但是运行时不包含schema。默认值是 *false*
immutable	此属性用于选择MBG是否会生成不可变的模型类。 这意味着这些类不会包含"setter"方法，而且通过构造参数接收类中所有字段的值。如果设置为true,将会忽略"constructorBased"属性， 强制生成通过参数化构造方法的模型类。此属性仅适用于MyBatis3，iBATIS2将会被忽略。默认值是 *false*
modelOnly	此属性用于选择MBG是否只会为表生成模型类。如果设置为true，那么就不会生成Java客户端类。 如果<sqlMapGenerator> 配置了，并且属性设置为true，那么MGB将会在该表的SQL映射XML中只生成结果映射元素。如果设置为true，这个值会覆盖<table>元素上所有的"enable*"属性，将不会生成任何CRUD方法。默认值是 *false*
rootClass	这个属性可以用来指定所有生成的Java模型类的基类。 如果表包含主键，MBG会将该值指定为主键对象的超类。 或者其他记录对象的超类。 这个值将会覆盖Java模型生成配置中配置的`rootClass`重要: 如果MBG可以加载基类，通常生成的属性不会覆盖和基类中完全匹配的属性。 完全匹配属性的定义如下
	属性名称完全相同
	属性的类型相同
	属性有一个"getter"方法
	属性有一个"setter"方法
	如果指定这个值，这个属性值应该是一个完全限定的类名(例如：com.mycompany.MyRootClass).
rootInterface	这个属性可以用来指定所有生成的DAO接口对象的父接口。 这个属性值会覆盖DAO生成器配置中配置的`rootInterface`属性。重要: MBG 不会校验接口是否存在或者合法。如果指定这个值，这个属性值应该是

	一个完全限定的接口名 (例如：com.mycompany.MyRootInterface).
runtimeCatalog	如果您指定了这个属性值，MBG会在生成的SQL中使用这个catalog，而不是前面配置的`catalog` 属性值。当您生成代码的catalog和运行时的catalog不一样时，这会非常有用。
runtimeSchema	如果您指定了这个属性值，MBG会在生成的SQL中使用这个schema，而不是前面配置的`schema` 属性值。当您生成代码的schema和运行时的schema不一样时，这会非常有用。
runtimeTableName	如果您指定了这个属性值，MBG会在生成的SQL中使用这个表名，而不是前面配置的`tableName` 属性值。当您想在Oracle中使用公共同义词生成对象时，这会非常有用。 在这种情况下，您将需要生成对象的同义词的指向您实际表。那就在此属性中指定同义词名称。在使用公共同义词的大多数情况下，您还需要设置`ignoreQualifiersAtRuntime`属性。
selectAllOrderByClause	这个属性可以用于指定将会加到 `selectAll` 方法中的order by语句。 这仅适用于您使用`MyBatis3Simple`目标运行环境时。 MBG会将您在这儿指定的任何属性值前面追加`order by`， 因此该属性赢仅包含列表中的列(例如 `ID1, ID2` 或 `ID1 desc, ID2 asc`)
useActualColumnNames	如果设置为true,那么MBG会使用从数据库元数据获取的列名作为生成的实体对象的属性。 如果为false(默认值)，MGB将会尝试将返回的名称转换为驼峰形式。 在这两种情况下，可以通过 <columnOverride> 元素显示指定，在这种情况下将会忽略这个（useActualColumnNames）属性。例如，假设一个表包含START*DATE*列，如果这个属性是*"true",* 那么*MBG*生成的属性名是`START_DATE` - 这意味着这个值的*getter*和*setter*方法将会是 `getSTART_DATE()` 和`setSTART_DATE()`. 如果这个属性值是*fasle,MBG*将会生成的属性名是 `startDate` - 这意味着这个值的*getter*和*setter*方法将会是 `getStartDate()` 和`setStartDate()`.默认值是 false_
useColumnIndexes	如果是true,MBG生成resultMaps的时候会使用列的索引,而不是结果中列名的顺序. 当表中的列名的区别只是大小写的时候,这会非常有用. 这个支持还会有一个轻微的性能优势.默认值是 *false*重要提示: 当目标运行环境是Mybatis版本3时不支持这个属性.
useCompoundPropertyNames	如果是true,那么MBG生成属性名的时候会将列名和列备注接起来. 这对于那些通过第四代语言自动生成列(例如:FLD22237),但是备注包含有用信息(例如:"customer id")的数据库来说很有用. 在这种情况下,MBG会生成属性名FLD2237*CustomerId.*默认值是 false_

示例

元素 元素

元素用来从内省数据库中选择一个表。选定的表会生成如下的这些对象： 一个 MyBatis/iBATIS 格式的SQL映射文件 一组根据表生成的"model"类: 一个和表的主键匹配的类(如果表有主键)。 一个包含了非主键字段和BLOB字段的类，如果存在

这个元素指定我们总是想为一个在MYSCHEMA schema上的叫做 MYTABLE 的表生成代码. 我们还想忽略表中一个叫"fred"的列,而且我想还想重写"BEG_DATA"列,以便生成的属性名是"startDate".

```
<table tableName="MYTABLE" schema="MYSCHEMA">
  <ignoreColumn column="fred"/>
  <columnOverride column="BEG_DATE" property="startDate"/>
</table>
```

使用生成的对象

MyBatis Generator (MBG) 生成这些类型的对象:

1. Java模型对象 (通常)
2. SQL映射文件 (iBATIS通常有, MyBatis经常有)
3. SQL映射文件 (可选的)
4. 一个在xxxByExample方法中使用的类。有关这些类的信息，请参阅下列网页：
    - Example类用法说明
    - 扩展Example类

这些独立的页面有这些对象的说明和用法。

JAVA实体对象

MyBatis Generator (MBG)根据数据库表字段生成Java类. MBG生成的是简单实体对象而非带业务逻辑的实体对象(详见设计理念介绍).MBG根据表特性和配置生成不同类型的实体对象.

MBG生成字段和方法时包含JavaDoc标签 `@mbggenerated` .运行Eclipse插件时,MBG会将每个Java对象中的JavaDoc标签删除或替换.其他地方不会发生变化. 在这种情况下,您在Java类中添加的字段和方法就不用担心丢失--不包括包含了JavaDoc `@mbggenerated` 标签部分内容.

离开Eclipse插件，您需要手动合并Java文件,但是使用JavaDoc的 `@mbggenerated` 标签用来了解删除以前版本的文件是否安全.

以下描述生成不同的实体对象.MBG根据<context>的 `defaultModelType` 属性(支持conditional、flat、hierarchical三个值)和<table>的 `modelType` (此属性用于覆盖默认的defaultModelType)属性配置生成不同的实体对象.

任何字段配置了<ignoreColumn>,自动生成时不会添加到Java对象中.

注意: 下面描述中"BLOB"指任何列的BLOB数据类型包括BLOB, CLOB, LONGVARCHAR, 和 LONGVARBINARY.

主键类

主键类在表中包含一个主键的字段属性. (MBG)根据表列明自动生成属性名称.自动生成的属性名称可以配置 `<columnOverride>` 属性覆盖.

类名默认情况是 `?TableName?Key` ,如果 `<table>` 配置了 `domainObjectName` 属性那么类名是 `?domainObjectName?Key` .

如果表中存在一个字段的主键将会生成hierarchical类型实体对象. 如果表中存在多列组合主键将会生成conditional类型实体对象.主键类是不会生成flat类型实体对象model.

记录类

主键类在表中不包含主键、BLOB的字段属性. 如果只有一个主键记录类将继承主键类. (MBG)根据表列明自动生成属性名称.自动生成的属性名称可以配置 `<columnOverride>` 属性覆盖.

类名默认情况是 `?TableName?` ,如果 `<table>` 配置了 `domainObjectName` 属性那么类名是 `?domainObjectName?` 配置的值.

如果表中不存在组合主键、BLOB列将生成hierarchical类型实体对象.如果表中不存一个字段的主键、BLOB列或只有一个主键或者一个BLOB列将生成conditional类型实体对象.记录类常常生成flat类型实体对象.

BLOB记录类

BLOB记录类在表中包含BLOB字段属性. 如果表中只存在一个字段该类将继承基础类, 或将继承主键类(注意:MBG不支持表中只包含BLBO列). (MBG)根据表列明自动生成属性名称.自动生成的属性名称可以配置 <columnOverride> 属性覆盖.

BLOB记录类调用 selectByPrimaryKey 或 selectByExampleWithBLOBs 方法返回BLOB字段值.

类名默认情况是 ?TableName?WithBLOBs ,如果 <table> 配置了 domainObjectName 属性那么类名是 ?domainObjectName?WithBLOBs .

如果表中存在一个BLOB列将生成hierarchical类型实体对象. 如果表中存在多个BLOB列将生成conditional类型实体对象.BLOB记录类是不会生成flat类型实体对象.

Example类

Example类用来处理MBG动态查询功能. Example类设置条件用于动态WHERE子句下列方法中:

- selectByExample
- selectByExampleWithBLOBs
- deleteByExample
- countByExample
- updateByExample

Example类不继承任何实体对象.

类名默认情况是 ?TableName?Example ,如果 <table> 配置了 domainObjectName 属性那么类名是 ?domainObjectName?Example .

如果方法被启动Example类将生成任何 *ByExample 方法.注意:如果表中有非常多的字段该类可能很大,但 DAO生成的XML是比较小的. 如果您不需要使用动态WHERE子句,您可以禁用生成这些方法.

详见Example类使用说明了解详情.

SQL映射文件

MyBatis Generator (MBG)生成SQL映射文件遵循MyBatis或iBATIS SQL Map DTD规范. 在表的基础上这些文件还包含了不同的标签和属性配置. MBG按照配置表生成SQL映射文件.表名就是SQL映射文件的命名空间(前提是数据库支持schema和catalog,主流数据库对schema和catalog都支持).MBG不会自动把 SQL映射文件到MyBatis/iBATIS配置文件中 - 您必须手动把 `xxxMapper.xml` 文件加入到配置文件中(或者您使用一个插件使MBG生成配置文件;Spring配置文件可以实现自动扫描 `xxxMapper.xml` 和接口).

每个自动生成的XML方法标签下都包含 `@mbggenerated` 的注释块.在运行的时候 `@mbggenerated` 注释块将删除和替换.其他部分内容将保持不变. 考虑到这点,不用担心在运行时丢失您新增内容--不包括 `@mbggenerated` 块内容.

以下部分描述了将要生成的元素.

注意:下面描述中"BLOB"指任何列的BLOB数据类型包括BLOB, CLOB, LONGVARCHAR, 和 LONGVARBINARY.

结果集

结果集用于数据库表列映射Java对象的属性. 结果集(和相应的查询语句)不包括如下情况:

- 任何列配置 `<ignoreColumn>` 属性将被会被忽略
- 任何BLOB字段(详见有关BLOBs字段相关映射)

对于列配置 `<columnOverride>` 属性将映射成配置的名称.没有配置使用的是默认的属性和JDBC类型.

对于自定义连接查询结果集继承一个结果集是一种非常常见的用法. 对于其他连接查询也想使用该结果集在MBG自动生成的时候需要配置前缀.详见<table>属性alias配置前缀,这样做法可以区分表中相同字段.

如果table配置enableSelectByExample、enableSelectByPrimaryKey属性为true,MBG会生成结果集,table默认这两个配置都是true,所以大部分情况下都会生成resultMap.

BLOB结果集

表中存在BLOB字段时MBG会自动生成一个BLOB结果集,BLOB结果集继承了除BLOB字段外的基础结果集. 我们提供不同的版本查询语句,来适应查询结果中是否需要BLOB字段.

BLOB结果集(和相应的查询语句)不包括如下情况:

- 任何列配置 `<ignoreColumn>` 属性将被会被忽略

对于列配置 `<columnOverride>` 属性将映射成配置的名称.没有配置使用的是默认的属性和JDBC类型.

对于自定义连接查询结果集继承一个结果集是一种非常常见的用法.对于其他连接查询也想使用该结果集,在MBG自动生成的时候需要配置前缀.详见<table> 性alias配置前缀,这样做法可以区分表中相同字段.

如果table存在BLOB字段且配置了enableSelectByExample、enableSelectByPrimaryKey属性,MBG会生成BLOB结果集,table默认这两个配置都是true所以大部分情况下都会生成resultMap.

Where条件SQL语句

"by example"方法重用Where条件.被重用的Where条件不包括BLOB字段.大部分表WHERE条件都不支持BLOB字段.

"by example"语句被配置将动态生成Where条件语句.

根据主键查询

select语句按照主键查询返回一行数据.如果表中存在BLOB字段,查询结果中会包含该字段.

如果table配置了enableSelectByExample、enableSelectByPrimaryKey属性为true,MBG会生成结果集,table默认这两个配置都是true所以大部分情况下都会生成resultMap.

根据条件查询

根据条件查询结果集和自动生成对象相匹配. 通过实现一个简单的"query by example"方法可以适用于不同的数据库查询.就算表中存在BLOB字段该查询结果集也不会返回该BLOB字段 (具体返回BLOB字段详见根据条件查询包含BLOB字段结果集部分描述).

重要:条件为空或者没有设置条件时,会将表中所有数据查询出来.

如果table配置了enableSelectByExample属性为true,MBG会生成Example查询.

根据条件查询包含BLOB字段(方法:selectByExampleWithBLOBs)

Example查询结果集和自动生成对象相匹配. 通过实现一个简单的"query by example"方法可以适用于不同的数据库查询.算表中存在BLOB字段该查询结果集将返回该BLOB字段.

重要: 查询条件为空或者没有设置查询条件时,会将表中所有数据查询出来.

如果table配置了enableSelectByExample属性为true,MBG会生成包含BLOB字段的Example查询.

插入

该方法可以插入表中所有字段(包括BLOB字段),但列配置了 `<ignoreColumn>` 属性将不会插入.

如果表有主键自动增长(自动增长列或序列),table子元素配置 `<generatedKey>` 属性,MBG会生成 `<selectKey>` 块语句,一般情况下都不需要配置使用数据库自动增长即可,除非表字段需要使用序列.

重要提示:与iBATIS2和MyBatis3插入的差异.差异如下:

运行	特性
iBATIS2	配置了<generatedKey>属性,插入的时候会返回新的自动增长值.没有配置<generatedKey>属性, 插入返回值为 `void` .
MyBatis3	插入方法将返回插入受影响的行数(通常是0或者1). 配置了<generatedKey>属性,自动增长值会被设置在参数对象中返回.

如果table配置了enableInsert属性为true,MBG会生成该方法.

选择性插入

该方法可以插入表中所有字段(包括BLOB字段),但列配置了 `<ignoreColumn>` 属性将不会插入.然而,参数对象 `为空` 的列将不会插入值.如果数据库列设置了默认值则取默认值,否则列值是空. 对于列不允许为 `空` 情况-则需要人为确保调用该方法时不为空列确保有值使插入不报错.重要:任何映射到java引用类型的字段都会被该方法插入.

如果表有主键自动增长(自动增长列或序列),table子元素配置 `<generatedKey>` 属性,MBG会生成 `<selectKey>` 块语句,一般情况下都不需要配置使用数据库自动增长即可,除非表字段需要使用序列.

重要提示:与iBATIS2和MyBatis3插入的差异.差异如下:

运行	特性
iBATIS2	配置了<generatedKey>属性,插入的时候会返回新的自动增长值.没有配置.没有配置<generatedKey> 属性, 插入返回值为 `void` .
MyBatis3	插入方法将返回插入受影响的行数(通常是0或者1). I配置了<generatedKey>属性,自动增长值会被设置在参数对象中返回.

如果table配置了enableInsert属性为true,MBG会生成该方法.

根据主键更新

该方法按照主键修改一行数据.该方法会修改表中所有字段,除非:

- 列配置了 `<ignoreColumn>` 属性
- BLOB字段(详见根据主键更新包含BLOB字段)

数据库表中存在主键且table配置了enableUpdateByPrimaryKey属性为true,MBG会生成该方法.

根据主键更新包含**BLOB**字段

该方法按照主键修改一行数据.该方法会修改表中所有字段(包括BLOB字段),除非:

- 列配置了 `<ignoreColumn>` 属性

数据库表中存在主键包含BLOB列且table配置了enableUpdateByPrimaryKey属性为true,MBG会生成该方法.

根据主键选择性更新

该方法按照主键修改一行数据.该方法会更新表中参数对象属性不为空的列.该方法可以被使用更新某些特定的列而不影响所有列数据. 重要**:** 任何映射到java引用类型的字段都会被该方法更新.

数据库表中存在主键且table配置了enableUpdateByPrimaryKey属性为true,MBG会生成该方法.

根据主键删除记录

该方法按照主键删除一行数据.

数据库表中存在主键且table配置了enableDeleteByPrimaryKey属性为true,MBG会生成该方法.

根据条件删除记录

该方法按照条件将删除一行或者多行数据.

重要**:**条件为空或者没有设置条件时, 会将表中所有记录删除.

如果table配置了enableDeleteByExample属性为true,MBG会生成该方法.

根据条件查询记录总数

该方法按照条件查询表中符合条件记录的记录总数.

重要**:**条件为空或者没有设置条件时,返回表中所有记录总数.

如果table配置了enableCountByExample属性为true,MBG会生成该方法.

根据条件更新记录

该方法按照条件更新表中符合条件记录.该方法会修改表中所有字段,除非:

- 列配置了 `<ignoreColumn>` 属性
- BLOB字段(详见根据条件修改记录包含BLOB字段)

重要**:**条件为空或者没有设置条件时, 会将表中所有记录更新.

如果table配置了enableUpdateByExample属性为true,MBG会生成该方法.

根据条件修改记录包含**BLOB**字段

该方法按照条件更新表中符合条件记录.该方法会修改表中所有字段(包括BLOB字段),除非:

- 列配置了 `<ignoreColumn>` 属性

重要**:**条件为空或者没有设置条件时, 会将表中所有记录更新.

如果表存在BLOB字段且table配置了enableUpdateByExample属性为true,MBG会生成该方法.

根据条件选择性更新记录

该方法按照条件更新表中符合条件记录.该方法会更新表中参数对象属性不为空的列.该方法可以被使用更新某些特定的列而不影响所有列数据.重要**:** 任何映射到java引用类型的字段都会被该方法更新.

重要**:**条件为空或者没有设置条件时, 会将表中所有记录更新.

如果table配置了enableUpdateByExample属性为true,MBG会生成该方法.

Java客户端对象

MyBatis Generator (MBG)生成几种类型的Java客户端对象.用Java的客户端对象与所产生的XML交互容易得多. 对于配置中的每个表，MBG生成一个或多个Java客户端对象.如MyBatis的3, 这些都接口映射.对于iBATIS的2.x中，这些都是DAO接口和实现类. 生成Java客户对象是可选的,由 `<javaClientGenerator>` 配置. MBGr可以产生以下的Java客户短对象类型:

- 如MyBatis 3.x:
  - XMLMAPPER - 用与支持MyBatis 3.x映射
- 如iBATIS 2.x:
  - IBATIS - 使用iBATIS DAO框架
  - SPRING - 与Spring框架联合使用
  - GENERIC-CI - 使用不依赖iBATIS数据映射
  - GENERIC-SI - 使用不依赖iBATIS数据映射

每个字段和方法产生包括JavaDoc标签 `@mbggenerated` .Eclipse插件运行时, 运行中每个字段和方法包括此Javadoc标记将被删除并替换.在类中的其他任何字段或方法将保持不变. 考虑到这一点,您可以不用担心加入的字段和方法丢失 -不包括在Javadoc标签 `@mbggenerated` 中内容.

离开Eclipse插件,您需要手动合并Java文件,但是使用JavaDoc的 `@mbggenerated` 标签用来了解删除以前版本的文件是否安全.

注意: 下面描述中"BLOB"指任何列的BLOB数据类型包括BLOB, CLOB, LONGVARCHAR, 和 LONGVARBINARY.

通用DAO方法

根据表的特性,以及配置选项,Java客户端自动生成如下方法:

- countByExample
- deleteByPrimaryKey
- deleteByExample
- insert
- insertSelective
- selectByPrimaryKey
- selectByExample
- selectByExampleWithBLOBs
- updateByPrimaryKey (否更新BLOB字段需要重写方法)
- updateByPrimaryKeySelective (只更新参数类非空字段)
- updateByExample (否更新BLOB字段需要重写方法)
- updateByExampleSelective (只更新参数类非空字段)

对于包含BLOB的表,MBG通过生成不同的对象和方法使您更容易使用BLOB字段,是否忽略它们,这取决于具体情况.

详见 Example类使用说明 中 `selectByExample` 方法使用.

XMLMAPPER 客户端 (MyBatis 3.x)

XMLMAPPER客户端是将接口方法映射到生成的XML映射文件中.例如,MBG自动生成的接口名为 `MyTableMapper` .您可以如下使用该接口:

```
  SqlSession sqlSession = sqlSessionFactory.openSession();

  try {
    MyTableMapper mapper = sqlSession.getMapper(MyTableMapper.cl
ass);
    List<MyTable> allRecords = mapper.selectByExample(null);
  } finally {
    sqlSession.close();
  }
```

有关如何创建实例的详细信息,请参阅MyBatis文档怎么创建 `sqlSessionFactory` 实例.

IBATIS DAOs (iBATIS 2.x)

iBATIS DAO依赖于iBATIS的DAO框架(iBATIS的一个可选部分 - 现在已经过时). 他们继承SqlMapDaoTemplate类和初始化DAOManager对象的实例,并且调用方法执行不同的语句.

MBG不会为您更新“dao.xml”文件 - 您必须添加手动相应的内容.

iBATIS DAO框架是一个非常基础的IoC容器,如果您尚未使用像Spring或PicoContainer的管理依赖关系的框架,iBATIS DAO框架可能会有用. 然而,该框架现在已经过时,我们建议您使用Spring.

SPRING DAOs (iBATIS 2.x)

SPRING DAO依赖于Spring框架.他们继承Spring的SqlMapClientDaoSupport类通过Spring容器构造DAO.

GENERIC-CI DAOs (iBATIS 2.x)

GENERIC-CI DAO调用iBATIS的SqlMapClient接口.该接口的实例是通过构造函数注入.

GENERIC-SI DAOs (iBATIS 2.x)

GENERIC-SI DAO调用iBATIS的SqlMapClient接口.该接口的实例是通过setter注入.

Example类使用说明

Example类指定如何构建一个动态的where子句. 表中的每个non-BLOB列可以被包括在where子句中. 例子是展示此类用法的最好方式.

Example类可以用来生成一个几乎无限的where子句.

Example类包含一个内部静态类 `Criteria` 包含一个用 `anded` 组合在where子句中的条件列表. Example类包含一个 `List<Criteria>` 属性,所有内部类Criteria中的子句会用 `ored` 组合在一起. 使用不同属性的 `Criteria` 类允许您生成无限类型的where子句.

创建 `Criteria` 对象 可以使用Example类中的 `createCriteria()` 或者 `or()` . 如果 `Criteria` 对象是用 `createCriteria()` 创建的，它会自动为 `List<Criteria>` 属性添加一个 `Criteria` 对象 - 这使得它更容易写一个简单的where子句， 如果您不需要 `or` 或者其他几个子句组合的话. 用 `or(Criteria criteria)` 方法创建 `Criteria` 对象, 方法里的 `criteria` 对象会被添加进 `Criteria` 对象的列表中.

重要 我们推荐您只使用 `or()` 方法创建 `Criteria` 对象. 我们相信这种方法使代码更有可读性.

简单查询

这个例子展示了如何用生成后的Example类去生成一个简单的where子句:

```
TestTableExample example = new TestTableExample();

example.createCriteria().andField1EqualTo(5);
```

作为另一种选择, 下面的方式也是可以的:

```
TestTableExample example = new TestTableExample();

example.or().andField1EqualTo(5);
```

在上面的例子中, 动态生成的where子句是:

```
where field1 = 5
```

复杂查询

下面的例子展示了如何用生成后的Example类去生成一个复杂的where子句 (用到了JSE 5.0 的泛型):

```
TestTableExample example = new TestTableExample();

example.or()
  .andField1EqualTo(5)
  .andField2IsNull();

example.or()
  .andField3NotEqualTo(9)
  .andField4IsNotNull();

List<Integer> field5Values = new ArrayList<Integer>();
field5Values.add(8);
field5Values.add(11);
field5Values.add(14);
field5Values.add(22);

example.or()
  .andField5In(field5Values);

example.or()
  .andField6Between(3, 7);
```

在上面的例子中, 动态生成的where子句是:

```
where (field1 = 5 and field2 is null)
   or (field3 <> 9 and field4 is not null)
   or (field5 in (8, 11, 14, 22))
   or (field6 between 3 and 7)
```

将会返回满足这些条件的记录结果.

去重复查询

您可以在所有的Example类中调用 `setDistinct(true)` 方法进行强制去重复查询.

Criteria类

`Criteria` 内部类的每个属性都包含 `andXXX` 方法，以及如下的标准的SQL查询方法:

- IS NULL - 指相关的列必须为NULL
- IS NOT NULL - 指相关的列必须不为NULL

- = (equal) - 指相关的列必须等于方法参数中的值
- <> (not equal) - 指相关的列必须不等于方法参数中的值
- > (greater than) - 指相关的列必须大于方法参数中的值
- >= (greater than or equal) - 指相关的列必须大于等于方法参数中的值
- < (less than) - 指相关的列必须小于于方法参数中的值
- <= (less than or equal) - 指相关的列必须小于等于方法参数中的值
- LIKE - 指相关的列必须 "like" 方法参数中的值. 这个方法不用必须加入 '%', 您必须设置方法参数中的值.
- NOT LIKE - 指相关的列必须 "not like" 方法参数中的值. 这个方法不用必须加入 '%', 您必须设置方法参数中的值.
- BETWEEN - 指相关的列必须在 "between" 方法参数中的两个值之间.
- NOT BETWEEN - 指相关的列必须不在 "not between" 方法参数中的两个值之间.
- IN - 指相关的列必须在传入的方法参数的list中.
- NOT IN - 指相关的列必须不在传入的方法参数的list中.

扩展Example类

在某些情况下可能需要扩展自动生成的example。您可能希望添加特定数据库查询条件(如Oracle ROWNUM支持),或添加除自动生成外的查询条件(如不区分大小写查询)。在这种情况下，您需要扩展自动生成example类来添加这些额外的查询条件。

一般原则

MyBatis Generator (MBG)一般情况下一个表名对应一个自动生成"example"类,除非您特殊配置。"example"类动态生成where条件被用于 `xxxByExample` 方法。

标准的"example"类包含只是标准SQL条件查询功能。在这种情况下,程序特定需要您添加额外条件查询功能。这可能需要添加非标准条件查询或在where条件中使用数据库特定功能。

自动生成"example"类中包含一个内部类实现where条件查询功能。内部类命名为 `GeneratedCriteria` 。MBG同时也生成了内部类 `Criteria` 继承了 `GeneratedCriteria` ,您可以使用它在example类中添加您想要的功能。Eclipse插件不会删除 `Criteria` 类新增代码(注：只有Eclipse插件,并且有注释和时间戳等要求才能自动合并),因此您无需担心新增代码丢失。

例如,有一个表叫CUSTOMER。通常,MBG生成一个名为 `CustomerExample` 的类。在 `CustomerExample` 类中添加额外功能,需要在 `CustomerExample.Criteria` 类中新增方法。

扩展vs插件

如果您经常扩展自动生成类,写一个插件来实现该功能比手工编写扩展类代码更方便。下面(标题"单条件参数")的插件类能够完成单参数查询 `org.mybatis.generator.plugins.CaseInsensitiveLikePlugin` 。

添加条件语句

MBG自动生成SQL在运行允许创建无限制where条件。为了完成这个,自动生成SQL支持四大类型条件语句。对应每种类型的SQL语句, `GeneratedCriteria` 内部有一个对应的方法用于添加一个动态的where条件。

1. 简单字符串替换

在使用这种类型的条件查询时不需要参数对象替换where条件中。例如:

`FIRST_NAME is null` `LAST_NAME is not null`

此条件语句 `GeneratedCriteria` 类方法是:

`addCriterion(String anyString)`

其中"anyString"是字符串替换where子句。该方法适合任何类型。

例如,您想使用SOUNDEX函数完成"类似"搜索功能。在MySQL中,条件语句应该是:

`SOUNDEX(FIRST_NAME) = SOUNDEX('frod')`

此种查询太复杂，可以考虑使用另外一个方法,这种简单字符串替换必须插入到where条件中。在内部类 `Criteria` 中添加如下方法:

```
public Criteria andFirstNameSoundsLike(String value) {
  StringBuffer sb = new StringBuffer("SOUNDEX(FIRST_NAME) = SOUN
DEX('");
  sb.append(value);
  sb.append("')");

  addCriterion(sb.toString());

  return this;
}
```

下面代码是在 `selectByExample` 方法中使用了刚才新增的方法:

```
CustomerExample example = new CustomerExample();
Criteria criteria = example.createCriteria();
criteria.andFirstNameSoundsLike("frod");
List results = selectByExample(example);
```

这种方法可以添加任何条件语句到where子句中。然而,由于需要保证不同数据类型的正确(最明显的日期、时间和时间戳),所以最好使用参数替换。同时, 这样操作暴露过多的方法会导致SQL注入问题。如果可能,我们建议使用下面列出的方法之一。

2. 单条件参数

使用这种类型作为条件语句,一个参数替换where条件。例如

`FIRST_NAME = ?` `LAST_NAME <> ?`

自动生成 `Criteria` 类条件方法如下:

`addCriterion(String anyString,Object anyObject,String propertyName)`

Where:

anyString

替换where条件参数子句,如:upper(FIRST_NAME) like

anyObject

条件值

propertyName

条件列名,用于排除潜在错误。

该方法用于单一参数where条件。

例如,假设您想特定列不区分大小写查询,在MySQL中查询条件如下:

`upper(FIRST_NAME) like ?`

此方法适合单个参数功能-一个参数一个参数值。将下面方法添加到 `ExtendedCriteria` 中:

```
public ExtendedCriteria andFirstNameLikeInsensitive(String value
) {
  addCriterion("upper(FIRST_NAME) like",
    value.toUpperCase(),"firstName");

  return this;
}
```

下面代码是在 `selectByExample` 方法中使用了刚才新增的功能:

```
ExtendedExample example = new ExtendedExample();
ExtendedCriteria criteria = (ExtendedCriteria) example.createCri
teria();
criteria.andFirstNameLikeInsensitive("fred%");
List results = selectByExample(example);
```

3. 列表条件

列表条件适用于where条件中多个值的情况。例如:

`FIRST_NAME IN (?,?,?)` `LAST_NAME NOT IN (?,?,?,?)`

由于包含了"in" and "not in"这样的标准查询条件,因此使用起来不太灵活。然而在 `Criteria` 类中您会发现有相应的方法,如下:

`addCriterion(String anyString,List listOfObjects,String propertyName`

Where:

anyString

替换where条件参数,如:FIRST_NAME IN

listOfObjects

list对象值替换条件值(在list前有一个开始的括号，一个结束括号在list后)。

propertyName

条件列名,用户排除潜在错误。

4. Between条件

Between条件参数适用于where条件特定的格式。例如:

`FIRST_NAME BETWEEN ? AND ?` `LAST_NAME NOT BETWEEN ? AND ?`

由于包含了"between" and "not between"这样的标准查询条件,因此使用起来不太灵活。然而在 `Criteria` 类中您会发现有相应的方法,如下:

```
addCriterion(String anyString,Object object1,Object object2,String p
```

Where:

anyString

替换where条件参数,如:FIRST_NAME BETWEEN

object1

替换where条件第一个参数值(object1后会自带一个"and"连接词)。

object2

替换where条件第二个参数值(object2前会自带一个"and"连接词)。

propertyName

条件列名,用户排除潜在错误。

使用注意事项

不同数据库使用注意事项。

- DB2
- MySql
- Oracle
- PostgreSQL

请让我们知道您所使用的数据库,我们很乐意将信息添加到本节以供将来参考。

DB2 使用注意事项

LONG VARCHAR 字段

默认情况下MyBatis将LONG VARCHAR字段映射为 `java.lang.String` 类型, 数据库字段映射为 `jdbcType="LONGVARCHAR"` 。这样映射会导致DB2在检索数据时发生错误。DB2 的LONG VARCHAR字段应该被映射为 `java.lang.String` 类型，数据库类型应该是 `jdbcType="VARCHAR"` 。为了解决该问题, 使用 `<columnOverride>` 配置如下:

```
  <table schema="DB2ADMIN" tableName="ALLTYPES" >
    <columnOverride column="LONG_VARCHAR_FIELD" javaType="java.lang.String" jdbcType="VARCHAR" />
  </table>
```

MySql 使用注意事项

无符号类型

MySql支持有符号，无符号，数字类型字段。这些不是JDBC类型，因此MyBatis 生成器不能自动转换这种类型的字段。JAVA数据类型都是有符号的， 当使用无符号类型时这就会导致缺失精度。可以使用 `<columnOverride>` 解决MySql无符号数值类型的字段。下面是一个如何处理无符号bigint字段类型例子:

```
<table tableName="ALLTYPES" >
  <columnOverride column="UNSIGNED_BIGINT_FIELD" javaType="java.lang.Object" jdbcType="LONG" />
</table>
```

您还必须自己强制将返回值转换为适当的类型(在上面这种情况下，需要转换为 `java.math.BigInteger`)。

Oracle 使用注意事项

公共同义词

如果您想要一个表生成对象公共同义词，您实际生成的对象执行真正的表名-然后在运行的时候修改表名。MyBatis自动生成器支持这样。

例如，假设有一个公共表名为"FRED"指向表"HR.EMPLOYEES"。下面的配置是基于HR.EMPLOYEES生成的对象，但运行的时候SQL只会指向FRED:

```
<table schema="HR" tableName="EMPLOYEES">
  <property name="ignoreQualifiersAtRuntime" value="true" />
  <property name="runtimeTableName" value="FRED" />
</table>
```

Oracle 长数据类型

Oracle的JDBC驱动会将LONG类型的列当成JDBC的LONGVARCHAR类型。由于Oracle驱动程序不支持，MyBatis将映射为CLOB。 因此在使用长数据类型时，您应该在生成器上配置一个列的重写将其映射到JDBC的VARCHAR类型。

PostgreSQL 使用注意事项

区分大小写

PostgreSQL所有的数据库标识符(表名,模式名,列明等)是区分大小写的。此外，PostgreSQL明显的偏好所有标识符使用小写字母。 如果您使用全部小写标识符的PostgreSQL，MyBatis Generator不需要任何额外的配置就会发现表并且编写正确的SQL。如果您使用混合或者大写模式，您需要适当配置MyBatis Generator:

- 当您的表或模式是大小写混合或全部大写时，使用 `delimitIdentifiers` 配置。
- 为每个大小写混合或大写的列可以指定 `<columnOverride>` 元素来指定列， 或者您可以指定 `delimitAllColumns` 属性来指定所有的列。

例子1:

```
  <table schema="HR" tableName="Employees"
      delimitIdentifiers="true" delimitAllColumns="true"/>
```

例子2...

```
  <table schema="HR" tableName="Employees" delimitIdentifiers="t
rue" >
    <columnOverride column="EmployeeId" delimitedColumnName="tru
e" />
    <columnOverride column="EmployeeName" delimitedColumnName="t
rue" />
  </table>
```

参考资料

本节收集了一些和MyBatis Generator有关的有用的技术主题。

- 从源码构建
- 扩展 MyBatis Generator
- 执行插件
- 日志信息

从源码构建

所有 MyBatis 代码生成器(MBG)发行版都包含源代码。 运行时的唯一依赖是 `ant.jar` - 为了圆满完成包含的Ant任务。 它是直接从源代码编译MBG - 只需要解压发行包并使用您喜欢的工具进行编译。

MBG发行包不包含编译期间的测试，或者编译期间必须部分的其他的类。 如果您想检查这些类，或者使用github上最新版本的源码编译MBG，那么按照下面的步骤操作：

1. MBG用Apache Maven构建。 首先您必须能运行maven。 如果您不熟悉Maven，这里有最简单的步骤 (针对 Windows):
   i. 从这里 http://maven.apache.org/ 下载Maven发行包。
   ii. 解压发行包到一个方便的位置。
   iii. 设置环境变量和PATH。示例: set JAVA_HOME=C:\JavaTools\jdk1.6.0_17 set M2_HOME=C:\JavaTools\apache-maven-3.0 set PATH=%PATH%;%M2_HOME%\bin;
2. 使用git从下面的位置检出代码 https://github.com/mybatis/generator
3. 在检出的目录下打开CMD执行 `mvn clean install` 命令，或者其他的Maven命令。

扩展MyBatis Generator

MyBatis Generator（MBG）是专为扩展性设计的。所有的代码生成使用Java和XML元素简单的DOM表示执行。

Java的DOM包含在 `org.mybatis.generator.api.dom.java` 包

该XML DOM包含在 `org.mybatis.generator.api.dom.xml` 包

这些类不足以实现每个可以想象到的代码生成的可能性，但是用于生成简单到中等复杂Java和XML代码相当有用的。

在配置文件中使用的选项，您可以提供自己的许多关键代码生成接口的实现。您也可以继承任何提供的实现来提供定制的行为。这个页面将介绍可用的公共API，并提供进一步调查源代码的指示。如果您对如何扩展MBG有任何的困难，您可以在支持的邮件列表mybatis-user@googlegroups.com 发送信息。

扩展与插入

虽然本网页上显示了许多不同的扩展点，在大多数情况下，这将是很容易通过使用插件扩展MBG。查阅开发插件的参考页面了解更多信息。

代码生成的主要扩展点是 `org.mybatis.generator.api.IntrospectedTable` 。实现一个代码生成器是一个有意义的任务，只有当您想要完全取代MBG的代码生成活动时才考虑。自从Abator的原始版本以来，很少出现有增强请求无法通过一个插件来处理的。

扩展点

MBG提供了许多不同的扩展点。以下部分列出扩展MBG的不同方法，并描述了可以实现与不同的扩展活动的类型。如果您需要一些了解不同的选择的帮助，您可以自由的在用户邮件列表上询问问题。

org.mybatis.generator.api.IntrospectedTable

`IntrospectedTable` 是一个可以扩展用于提供和MBG供给的版本不同的代码生成的抽象类。这种实现的一个很好的例子是基于一个FreeMarker的或Velocity模板的实现。在大多数其他情况下，编写插件是一个更好途径。

如果您选择扩展这个类，您必须提供代码来生成Java和XML文件。您可以选择生成的这些文件的技术。 `IntrospectedTable` 基础类持有的一个 `org.mybatis.generator.internal.rules.Rules` 实例可查询许多用于代码生成的规则。

MBG提供反射表的几种实现。实现基于 `<context>` 元素的 `targetRuntime` 属性值。在大多数情况下，继承下面的一个内置的扩展比从头开始创建一个实现会容易很多。下表显示了内置的实现：

TargetRuntime	实现
MyBatis3（默认）	`org.mybatis.generator.codegen.mybatis3.Introspecte`
Ibatis2Java2	`org.mybatis.generator.codegen.ibatis2.Introspected`
Ibatis2Java5	`org.mybatis.generator.codegen.ibatis2.Introspected`

如果您选择实现此扩展点，指定 `<context>` 元素的 `targetRuntime` 属性为您的实现类的完全限定类名。

org.mybatis.generator.api.IntrospectedColumn

`IntrospectedColumn` 是一个包含列的信息的类，因为它是一个从数据库返回的元数据类。在某些罕见的情况下，可能需要重写这个类来提供自己的 - 特别是如果您创建了一套新的代码生成器。

如果您选择实施此扩展点，指定 `<context>` 元素的 `introspectedColumnImpl` 属性为您实现类的完全限定类名。

org.mybatis.generator.api.JavaTypeResolver

MBG在内省数据库时调用这个接口的方法将JDBC类型映射到Java类型。这个接口的默认实现 `org.mybatis.generator.internal.types.JavaTypeResolverDefaultImpl` 。您可以提供自己的实现，并且默认的实现已被设计为可扩展。

提供您自己的实现，在XML配置中指定全限定的类名：

```
    <javaTypeResolver type="mypackage.MyImplementation“> ...</ javaTypeResolver>
```

org.mybatis.generator.api.ShellCallback

MBG调用这个接口的方法来执行功能，它不能自己做。最重要的功能是：

- 翻译 项??目/包 到一个目录结构
- 合并事件中的Java源文件同名的现有的Java文件/程序包是否存在。

这个接口的默认实现 `org.mybatis.generator.internal.DefaultShellCallback` 。默认实现简单的将项目和包连接在一起，如果需要则创建必要的包目录。默认的实现不支持Java文件的合并，并将覆盖或忽略文件。

您可以提供自己的实现。如果您想将MBG集成到一些其他的环境中，这将是最重要的一个需要编写的类。例如，Eclipse插件提供的实现在Eclipse环境中运行时，它支持Java文件的合并。

提供您自己的实现，提供有关构造函数的接口 `org.mybatis.generator.api.MyBatisGenerator` 的实例对象。这个不能通过XML来配置。如果您提供自己的实现此接口的话，我们假设您也提供了一些附加的代码（如一个新的Ant任务）来运行您的实现。

org.mybatis.generator.api.ProgressCallback

MBG调用这个接口的方法在文件的生成（长时间运行的进程）中报告进度。这个接口的默认实现 `org.mybatis.generator.internal.NullProgressCallback` 它只是忽略所有的进度消息。您可以提供此接口的实现，支持进度通知和取消代码生成。

当集成MBG到其他IDE环境中实现此接口将是重要的。Eclipse的插件提供此接口挂接到Eclipse的进度通知系统的实现。

提供您自己的实现，在 `org.mybatis.generator.api.MyBatisGenerator.generate()` 的一个方法调用上提供其中一个接口的实例。这个不能通过XML来配置。同样，我们假设，如果您提供自己的实现此接口，那么您也提供了一些附加代码（如新的Ant任务或IDE集成）来运行您的实现。

开发插件

插件可用于修改或添加到由MyBatis Generator 生成的对象。插件必须实现 `org.mybatis.generator.api.Plugin` 接口 。插件接??口包含许多在代码生成过程的不同阶段被调用的方法。任何特定的插件通常不需要实现整个接口。因此，大多数插件应扩展适配器类 `org.mybatis.generator.api.PluginAdapter` 。适配器类提供基本的插件支持，并为大多数的接口方法（类似于Swing适配器类）提供了空操作的方法。

MyBatis Generator 提供了几个插件（都在包 `org.mybatis.generator.plugins` 下）。所提供的插件展示了不同类型的可以通过插件完成的任务。插件的源代码是可以下载的，也可以在线浏览这里 。

插件的生命周期

插件有一个生命周期。插件在代码生成过程的初始化期间创建并且在这个过程中的不同阶段被调用。下面的列表显示了插件的基本生命周期：

1. 插件通过默认的构造函数创建
2. `setContext` 方法被调用
3. `setProperties` 方法被调用
4. `validate` 方法被调用。如果该方法返回 `false` ，那么插件中的其他方法都不会再被调用。
5. 对于配置中的每个表：
   i. `initialized` 方法被调用
   ii. Java客户端的方法：^{1,2}
      i. `clientXXXMethodGenerated(Method, TopLevelClass, Introspe`
      - 当Java客户端实现类生成的时候这些方法被调用.
      ii. `clientXXXMethodGenerated(Method, Interface, Introspected`
      -当Java客户端接口生成的时候这些方法被调用。
      iii. `clientGenerated(Interface, TopLevelClass, IntrospectedTa`
      方法被调用
   iii. 模型方法：¹
      i. `modelFieldGenerated` , `modelGetterMethodGenerated` , `modelSetterMethodGenerated` for each field in the class
      ii. `modelExampleClassGenerated(TopLevelClass, IntrospectedTa`
      iii. `modelPrimaryKeyClassGenerated(TopLevelClass, Introspecte`
      iv. `modelBaseRecordClassGenerated(TopLevelClass, Introspecte`
      v. `modelRecordWithBLOBsClassGenerated(TopLevelClass, Intros`
   iv. SQL映射方法：¹
      i. `sqlMapXXXElementGenerated(XmlElement, IntrospectedTable)`
      - 当生成SQL映射的每个元素的时候这些方法被调用
      ii. `sqlMapDocumentGenerated(Document, IntrospectedTable)`
      iii. `sqlMapDocument(GeneratedXmlFile, IntrospectedTable)`

    v. `contextGenerateAdditionalJavaFiles(IntrospectedTable)` 方法被调用
    vi. `contextGenerateAdditionalXmlFiles(IntrospectedTable)` 方法被调用
6. `contextGenerateAdditionalJavaFiles()` 方法被调用
7. `contextGenerateAdditionalXmlFiles()` 方法被调用

注意事项: ¹ -这些方法将被包装的代码生成器调用。如果您提供一个自定义的代码生成器，那么这些方法将仅在自定义代码生成调用它们时调用。 ² -Java客户端的方法只有当配置Java客户端生成器的时候会被调用。

编写插件

实现一个插件，最好的办法是扩展 `org.mybatis.generator.api.PluginAdapter` 类并只需要覆盖您需要的插件方法。

插件接口方法可用于修改默认生成的代码，或添加其他生成的代码。Examples of things 可以通过插件实现的是：

- 给生成的方法添加自定义注解
- 给生成的类添加其他方法
- 给生成的XML文件添加其他元素
- 生成额外的Java文件
- 生成额外的XML文件

`contextXXX` 方法总是会被调用。其他方法通过包代码生成器调用 - 且仅当表中的一个规则将会生成一个特定元素时。例如，如果该表没有主键， `modelPrimaryKeyClassGenerated(TopLevelClass, IntrospectedTable)` 方法将不会被调用。

方法返回一个 `boolean` 可用于绕过生成代码。如果任意这些方法返回 `false` ，则相关的项目不会被添加到生成的代码中。如果配置了多个插件，那么第一个插件从方法返回false将导致MyBatis Generator停止调用所有其他插件的方法。

如果您有一个关于插件的想法，可以自由的在用户列表问有关的问题。我们在这里提供帮助！

日志信息

MyBatis Generator（MBG）报告日志记录的几种不同的方式：

- MBG每次运行的时候会记录并且显示警告信息。 这些消息的目的是告诉用户需要注意可能存在的重要事件，也可能不是。 例如文件正在被覆盖，或者非致命的配置错误等等。 无论配置和命令行参数怎么配置，警告总是会显示。
- MBG生成器每次运行的时候可能会或者不会显示进度信息。 这些信息用来告诉用户代码生成的器的进度信息。 这些信息默认不显示，但是可以通过指定 `-verbose` 命令行参数进行显示。 或者您通过Ant任务运行MBG，您可以通过设置 `verbose` 属性为 `true` , 这样Ant就会运行在verbose模式。
- 最后，MBG将生成跟踪(记录)详细的调试消息。本页说明如何启用这些语句。

一般来说，MBG不会重复消息。所以MBG生成警告的时候这个警告通常还没记录。 在某些情况下需要启用日志记录以及要求MBG记录详细进度消息的时候可能会有用。 这可能会产生大量的输出，但是仍然会提供一个MBG运行期间完整的内部事件描述。

如果Log4J在运行时的类路径中，MBG就会使用Apache Log4J进行记录。 查阅http://logging.apache.org/log4j/ 获取有关Log4J的详细信息。如果Log4J不在运行时的类路径中，MBG会使用标准的Java日志记录。

即使运行时类路径中存在Log4J，您仍然想强制使用标准的Java日志记录， 您可以在命令行参数指定 `-forceJavaLogging` ，或者当您通过Java执行MBG时使用执行下面的方法：

```
org.mybatis.generator.logging.LogFactory.forceJavaLogging();
```

重要: 您必须将上面的方法放在任何MBG代码之前。

提供备用的实现

如果您喜欢使用不同的日志实现比Log4J的或标准的Java日志记录，您可能会提供一个备用实施的关键日志界面如下：

1. 创建 `org.mybatis.generator.logging.Log` 接口的实现，实现您选择的日志实现的关键日志方法。
2. 创建 `org.mybatis.generator.logging.AbstractLogFactory` 接口的实现，将返回您 `Log` 实现的实例。
3. 配置MBG通过调用方法来使用新的LogFactory `org.mybatis.generator.logging.LogFactory.setLogFactory(AbstractL` 并提供您的实例 `AbstractLogFactory` 实现。

配置Log4J日志

下面是一个log4j配置文件示例：

```
# Set root logger
log4j.rootLogger=INFO, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r %-5p %c - %m%n

# MBG logging configuration...
log4j.logger.org.mybatis.generator=DEBUG
```

这个文件指示Log4J将MBG所有调试信息输出到控制台。要使用此文件：

1. 运行时类路径的根目录创建一个名为 `log4j.properties` 的文件
2. 将上面的条目复制到新文件
3. 运行MBG时，Log4J的JAR文件也需要在运行时类路径下。

您应该可以在控制台中看到很多日志信息。

如果您喜欢，您也可以配置log4j任何其他支持的方法。

配置Java日志记录

下面是一个Java日志记录配置的示例文件:

```
# 指定root logger要创建的处理程序
# (所有的loggers都是root logger的孩子)
handlers = java.util.logging.ConsoleHandler

# 为 root logger 设置默认日志记录级别
.level = INFO

# Set the default logging level for new ConsoleHandler instances
java.util.logging.ConsoleHandler.level = ALL

# Set the default formatter for new ConsoleHandler instances
java.util.logging.ConsoleHandler.formatter = java.util.logging.S
impleFormatter

# Set the default logging level for the logger named org.mybatis
.generator
org.mybatis.generator.level = FINE
```

这个文件将指导Java写的所有MBG调试消息到控制台。要使用此文件：

1. 创建一个名为 `logging.properties` （或者您喜欢的任何文件名 ??）的文

件。该文件可以在文件系统中存在的任何地方（例如，在一个 `\temp` 目录）。
2. 将上述条目复制到新文件
3. 运行MBG时带上这个VM参数： `-Djava.util.logging.config.file=\temp\logging.properties` （指定您使用的实际的文件名 ??和目录）

在控制台中您将看到许多日志信息。

如果您喜欢，您还可以配置Java日志记录为其他支持的方法。

提供的插件

随着MyBatis生成器（MBG）的使用量的增加，我们发现越来越有用以通过插件增加功能，而不是添加到基本代码发生器的复杂性。插件是一个模块化的，易于理解的机制，继承MBG。有关编写一个插件的更多信息，请参阅实现插件 。有关配置插件的信息，请参阅<plugin>

所提供的插件都在 `org.mybatis.generator.plugins` 包。所提供的插件展示不同类型的可以完成与MBG插件任务。插件的源代码可以和MBG一起下载，也可以在线浏览这里 。

org.mybatis.generator.plugins.CachePlugin

这个插件在生成的SQL映射中增加了一个<cache>元素。这个插件仅用于MyBatis3目标运行时环境。

这个插件接受下列属性。都是可选的，并且，如果指定，则值将被直接传递到相应的属性生成的<cache>元素。

- cache_eviction
- cache_flushInterval
- cache_readOnly
- cache_size
- cache_type

所有属性都可以通过指定<table>元素的属性来覆盖。

org.mybatis.generator.plugins.CaseInsensitiveLikePlugin

该插件给Example类添加方法（实际上是给Criteria内部类）来支持不区分大小写的LIKE搜索。这表明通过插件给Example类添加功能，而不是扩展这个类。

org.mybatis.generator.plugins.EqualsHashCodePlugin

这个插件给由MBG生成的Java模型对象增加了 `equals` 和 `hashCode` 方法。

通过这个类生成的 `equals` 方法，在大多数情况下是正确的，但如果您已经指定了rootClass可能是不正确的 - 因为我们的equals方法只检查它知道的字段。

org.mybatis.generator.plugins.MapperConfigPlugin

这个插件生成包含对MBG所生成的XML映射文件的框架MapperConfig.xml文件。此文件可用于配置的MyBatis 3.X环境。

这个插件接受三个属性：

- `fileName` ??（可选的）生成的文件的名称。如果没有指定，默认为“MapperConfig.xml”。
- `targetPackage` （必须的）生成文件放置的包名。指定像“com.mycompany.sql”这样的值。
- `targetProject` （必须的）该文件应放置在的项目名称。

注： `targetPackage` 和 `targetProject` 遵循sqlMapGenerator配置元素上 `targetPackage` 和 `targetProject` 的相同的规则。

org.mybatis.generator.plugins.RenameExampleClassPlugin

这个插件通过重命名由MBG生成的Example类的方法演示 `initialized` 方法的用法。

这个插件接受两个属性：

- `searchString` （必须的），用于搜索默认生成的Example的名称的正则表达式。
- `replaceString` （必须的）插在匹配 `searchString` 位置的字符串。

例如，从xxxExample重命名生成的例子类xxxCriteria，指定 `实例美元searchString` 和 `标准replaceString`

org.mybatis.generator.plugins.RowBoundsPlugin

这个插件将添加一个新版本 `selectByExample` 方法接受 `RowBounds` 参数。这支持的MyBatis RowBounds函数，其中一个返回的结果列表可以在长度受到限制，并且开始位置可以被指定。这可以是在分页应用中是有用的。

这个插件仅适用于MyBatis3目标运行时环境。

org.mybatis.generator.plugins.SerializablePlugin

这个插件给由MBG生成的Javas添加了 `java.io.Serializable` 标记接口。这个插件给实体类增加了 `serialVersionUID` 字段。

重要提示：这是一个简单的实现java.io.Serializable并且不会尝试做任何版本的类。

这个插件接受两个属性：

- `addGWTInterface` （可选的）True/False.。如果为true，插件将给实体对象增加谷歌Web工具包（GWT）的 `IsSerializable` 接口。默认值是false。
- `suppressJavaInterface` （必须的）True/False。如果为true，插件将不添加 `java.io.Serializable` 接口。这是对于其中对象应该是可序列化的GWT，但不是严格意义上的Java的场景。默认值是false。

org.mybatis.generator.plugins.SqlMapConfigPlugin

这个插件生成包含对MBG所产生的SqlMap.xml文件的SqlMapConfig.xml框架文件。此文件可用于配置的iBATIS 2.x的环境。

这个插件接受三个属性：

- `fileName` ??（可选的）所产生的文件的名称。如果没有指定，默认值是“SqlMapConfig.xml”。
- `targetPackage` （必须的）放置该文件的包名。指定像“com.mycompany.sql”这样的值。
- `targetProject（` 必须的）放置该文件的项目的名称。

注： `targetPackage` 和 `targetProject` 遵循和sqlMapGenerator配置元素上的 `targetPackage` 和 `targetProject` 相同的规则。

org.mybatis.generator.plugins.ToStringPlugin

该插件给实体类添加 `toString()` 方法。

org.mybatis.generator.plugins.VirtualPrimaryKeyPlugin

这个插件可用于指定作为主键的列，即使它们没有在数据库中被定义为主键列。这是在数据库表没有定义主的情况下非常有用。通常情况下，如果没有主键，MBG将产生一组非常有限的方法。这个插件可以用来启用生成的完整的MBG方法。

要使用该插件，添加属性“virtualKeyColumns”到您的<table>配置，设置值为应被视做主键的用逗号或空格分隔的列名列表。列名必须和数据库（通常全部大写）返回的列名完全匹配。例如：

```
<table tableName="foo">
  <property name="virtualKeyColumns" value="ID1, ID2" />
</table>
```

设计理念

此工具可能会引发一些哲理性问题， 因为该工具是更侧重于数据库表，而不是实体类。 我们将采取几个段落来谈论这种做法。

首先，这个工具做什么么。我们不是在做任何有关项目应该或不应该是结构化这种说法的。 在一般情况下我们支持富实体类， 但创建一个富实体类和是否应该坚持这种模式完全不是一码事。

如果您特定的设计理念是实体类驱动所有的决策，数据库设计是屈从于实体类， 那么这个工具 - 和MyBatis本身 - 可能不适合您的应用程序。 在这种情况下，我们建议您认真考虑 Hibernate - 他可能更贴近您的应用程序和设计理念。

但并非所有的项目都适合这种模式。很少有真正的企业级应用程序会这么做。MyBatis对那些将数据库表设计和实体对象设计看做一样的项目有很大的帮助。MyBatis不是对象关系映射，并且不会尝试去持久化对象。所以他着眼于应用开发人员编写SQL和数据库表进行交互。

在大型或企业级的项目，其中的许多因素是很常见:

- 数据库设计往往是从 OO 实体设计一个单独的功能（有单独的管理）
- 数据库设计者没有 OO 工具 （比如继承），所以他们不从 OO 考虑。
- 应用程序设计人员并不能完全控制数据库表的最后形式。 例如，在应用中适合存在于一个对象的数据，在数据库中可能被分成几个表。
- 数据库设计最终和OO设计的完全不同，从而导致表和对象之间的明显不能匹配。

从这些主要的指标来看，对您的应用来说 MyBatis 是一个非常不错的候选工具，MyBatis Generator 可以在这类项目起到重要的影响。 那么，在这种情况下应该如何使用MyBatis?

数据访问对象(DAO)模式仍然是一种主要的模式。MyBatis Generator可以生成一组基本的和每个表对应的对象。 生成的代码是和事务无关的。这意味着如果在一个事务中涉及多个表参与，可以很容易的扩展生成的代码。 或者您可以创建另一个DAO（或服务方法），更紧密地匹配实体对象的持久性需求，并可以再一个事务中使用一个或多个生成的对象。

举个例子，考虑一个典型的 `Order` 对象，典型的标题和细节的问题。 在某些环境中这种对象将保存到至少四个表 - 两个关键表、"标题"表和"详细信息"表 (再次声明，我们不做任何种类的关于这是否"正确"的语句设计，只在陈述一个事实)。 您的应用程序还应与 `Order` 对象进行交互, 并且可能在某个地方（在OrderDAO 或者在一个Service对象）存在一个 `saveOrder(Order order)` 方法 该方法将与所涉及的4个表生成的代码进行交互。

在这种情况下，代码生成器给我们带来什么? 有以下几种:

- 重用（复用） - 很可能是一些表将需要从不同的DAO或者Service方法访问。为每个表创建一个 DAO 可以促进重用和应用程序内的一致性。

- 数据库抽象 - 服务层通常在应用程序中定义持久性。这些方法可以很快稳定。随着数据库设计的发展：
  1. 代码可以很快的随着表的改变而重新生成
  2. Service方法可以根据需要修改
  3. 更高层的应用程序保持不变
- 开发者的效率 - 生成基于表的DAO是很快速、可重复和无差错的。 开发人员可以专注于对象持久化和可能需要的复杂的联接查询。
- 更少的缺陷 - 因为所有应用程序中最繁琐和最容易出错的部分（获得SQL匹配的对象）是自动化的。

Velocity 中文文档

1. 关于

Velocity 用户指南旨在帮助页面设计者和内容提供者了解Velocity 和其简单而又强大的脚本语言（Velocity Template Language (VTL)）。本指南中有很多示例展示了用Velocity来讲动态内容嵌入到网站之中，但是所有的VTL examples 都同演示用于所有的页面和模版。

感谢选择Velocity!

2. 什么是Velocity?

Velocity 是一个基于Java的模版引擎。它允许web 页面设计者引用JAVA代码预定义的方法。Web 设计者可以根据MVC模式和JAVA程序员并行工作，这意味着Web设计者可以单独专注于设计良好的站点，而程序员则可单独专注于编写底层代码。Velocity 将Java 代码从web页面中分离出来，使站点在长时间运行后仍然具有很好的可维护性，并提供了一个除JSP和PHP之外的可行的被选方案。

Velocity可用来从模板产生web 页面，SQL, PostScript以及其他输出。他也可用于一个独立的程序以产生源代码和报告，或者作为其它系统的一个集成组件。这个项目完成后，Velocity将为Turbine web 应用程序框架提供模板服务。Velocity+Turbine 方案提供的模板服务将允许web 应用按真正的mvc模式进行开发。

3. Velocity 可以做什么?

3.1. Mud Store 示例

假设你是一个专门销售泥浆（MUD）的在线商店的页面设计者。我们称他为"The Online Mud Store"。生意很好。客户订购各种各样的类型和数量的泥浆。他们使用他们的用户名和密码登陆到商店中来，就可以浏览他们的订货和购买其他东西。现在，赤土陶泥正在促销，这是一种很常用的泥巴。一少部分顾客很有规律的购买一种亮红土Bright Red Mud，这也是促销产品，但是不太常用，因此被移到页面的边缘。所有顾客的信息都在数据库中被跟踪，因此有一天问题出现了：为什么不使用Velocity来定位目标客户，这些客户对某种类型的产品特别感兴趣？

Velocity 使针对访问者个性的WEB页面客户化（个性化）非常容易。作为一个在线泥巴商店的站点设计者，以想在客户以登陆进展点后就看到它们想看的页面。

你遇到你公司的软件工程师，每个人都认为$customer 将保持当前登陆进入的客户信息，而$mudsOnSpecial 将士当前所有促销的泥巴。$flogger 对象包含有助于促销的方法。对于当前的任务，让我们仅关注这三个问题。记住，你不需要担心软件工程师如何从数据库中取得顾客信息，但你必须知道他们可以。这样可以使你专注于你的工作而软件工程师则忙于他们自己的工作。

你可以在你的页面中嵌入如下的VTL语句：

```
<HTML>
<BODY>
Hello $customer.Name!
<table>
#foreach( $mud in $mudsOnSpecial )
   #if ( $customer.hasPurchased($mud) )
      <tr>
        <td>
          $flogger.getPromo( $mud )
        </td>
      </tr>
   #end
#end
</table>
```

foreach语句的细节将进一步细说，但重要的是这个短小的脚本居然可以在你的站点上运行。当有一个倾向于亮红土的顾客登陆进来时，亮红土正在促销，这就是这个顾客所看到的，并且促销显示非常显著。如果另外一个长期购买赤陶土的顾客登陆进来，赤陶土促销的提示信息则应该在前面中间位置。Velocity是非常灵活的，受限的只是你的创造力。

写在VTL参考文档中的是其他Velocity 元素，他们一起给你很强大的能力和灵活性以创建很好的站点。待你更加了解这些元素，就可以开始释放Velocity的强大动力。

4. Velocity模板语言(VTL): 介绍

Velocity模板语言(VTL)旨在为Web页面结合动态内容提供最容易、简单和简洁的方法。即使有一点或者没有编程经验的页面设计者也可以很快能为页面提供动态内容。

VTL 使用引用（references）来将动态内容嵌入web页面，每个变量就是某一个类型的引用。变量实际上是一个可以调用定义在java代码中的内容的引用，或者它可以从页面内的VTL语句得出自身的值。下面是一个例子，说明可以嵌入到HTML文档中的VTL语句。

```
#set( $a = "Velocity" )
```

这个VTL语句，就像所有的VTL语句一样，以#字符开始，并跟着一个指令set。当一个在线访问这请求页面时， Velocity 模伴引擎在页面内搜索所有#字符，然后决定是哪一个标记了VTL语句的开始，哪个标记不需要VTL做什么动作。

#字符后面紧跟一个指令set.。set指令使用一个括在括号内的表达式---一个等式将一个值指派给一个变量。变量在等号的左边而值在等号的右边。

在上面的示例中，变量是$a值是Velocity。这个变量就象其他引用一样，以一个$字符开始。值通常在引号之中，对Velocity来说一般没有类型冲突的问题，因为只有字符串 (基于文本的信息)可以传递给变量。

下面的主要规则可能有助于理解Velocity 是如何工作的：引用以$开头用于取得什么东西，而指令以#开始用于做什么事情。

在上面的例子中，#set用于将一个值指派给一个变量。而变量$a则可以用来在模板中输出"Velocity" 。

5. Hello Velocity World!

一旦一个值被赋给一个变量，便可以在HTML中随处引用它。在下面的示例中，先给变量$foo赋值然后引用它。

```
<html>
<body>
#set( $foo = "Velocity" )
Hello $foo World!
</body>
<html>
```

这个页面的结果是输出"Hello Velocity World!"。

为了使包含VTL 指令的语句具有可读性，我们鼓励每个VTL语句在一个新行开始，虽然并不一定要这样做。set将随后深入解释。

6. 注释

可以用注释加入描述性文本，他们并不在模板引擎中输出。注释可以有助于你的记忆或者想其他人解释你的VTL语句正在做什么。

```
## This is a single line comment.
```

单行注释以##开始，并在本行结束。如果需要加入多行注释，并不需要加入很多的单行注释。多行注释，以#*开始并以*#结束可以处理这种情况。

```
This is text that is outside the multi-line comment.
Online visitors can see it.

#*
 Thus begins a multi-line comment. Online visitors won't
 see this text because the Velocity Templating Engine will
 ignore it.
*#

Here is text outside the multi-line comment; it is visible.
```

下面事一些例子说明单行注释和多行注释如何工作。

```
This text is visible. ## This text is not.
This text is visible.
This text is visible. #* This text, as part of a multi-line comm
ent,
is not visible. This text is not visible; it is also part of the
multi-line comment. This text still not visible. *# This text is
 outside
the comment, so it is visible.
## This text is not visible.
```

还有第三种注释， VTL 注释块，可以用来存储诸如文档作者、版本信息等。

```
#**
This is a VTL comment block and
may be used to store such information
as the document author and versioning
information:
@author
@version 5
*#
```

7. 引用

VTL中有三种类型的引用：变量，属性和方法。作为使用VTL的设计者，你和你的工程师必须在饮用的特定命名上取得一致，以便在你的模板中正确的使用他们。

有关引用的所有参数都处理为字符串对象。Everything coming to and from a reference is treated as a String object. 假如有一个对象表示$foo (比如说是整型对象)，Velocity 将调用其toString() 方法来将此对象转换为一个字符串。

7.1. 变量Variables

变量的简略标记是有一个前导"$"字符后跟一个 VTL 标识符（Identifier.）组成。一个VTL 标识符必须以一个字母开始(a .. z或 A .. Z)。剩下的字符将由以下类型的字符组成：

- 字母 (a .. z, A .. Z)
- 数字 (0 .. 9)
- 连字符("-")
- 下划线 ("_")

下面是一些有效的变量引用:

```
$foo
$mudSlinger
$mud-slinger
$mud_slinger
$mudSlinger1
```

当VTL 引用一个变量时，比如$foo，变量可以从模板的set指令取得值，也可以从

Java 代码中取得。例如，如果Java 变量$foo在模板被请求的时候具有值bar，则bar将替换页面中的所有$foo的实例。或者，如果包含下面的语句：

```
#set( $foo = "bar" )
```

紧跟指令后的所有$foo的实例的输出将会一样值。

7.2. 属性

VTL引用的第二种元素是属性，而属性具有独特的格式。属性的简略标记识前导符$后跟一个VTL 标识符，在后跟一个点号(".")最后又是一个VTL 标识符。这是一些有效的示例：

```
$customer.Address
$purchase.Total
```

请看第一个例子，$customer.Address.。他有两种意思。它可以意味着，查询由customer标是的哈希表并按关键字Address返回值。但是$customer.Address也可能引用一个方法（下述，$customer.Address可能是$customer.getAddress().的缩写。当一个页面被请求时，Velocity 将决定这两种可能到底是哪一个，然后返回相应的值。

7.3. 方法

方法在JAVA代码中定义，并作一些有用的事情，比如运行一个计算器或者作出一个决定。方法是实际上也是引用，由前导符"$"后跟一个VTL 标识符，后跟一个VTL 方法体（Method Body）。 VTL 方法体由一个VTL 标识符后跟一个左括号，再跟可选的参数列表，最后是右括号。下面是一些有效的方法示例：

```
$customer.getAddress()
$purchase.getTotal()
$page.setTitle( "My Home Page" )
$person.setAttributes( ["Strange", "Weird", "Excited"] )
```

前面两个例子-- $customer.getAddress()和$purchase.getTotal() – 看起来有点象上面属性一节中所用的样子，$customer.Address和$purchase.Total.。如果你想这些例子在某些方面相关，那你就对了。

VTL 属性可以为VTL方法用作简略标记。属性$customer.Address具有和方法$customer.getAddress() 完全一样的效果。属性和方法的主要不同点是方法中可以添加参数列表。

简略标记可以用在下面的方法中：

```
sun.getPlanets()
$annelid.getDirt()
$album.getPhoto()
```

我们或许希望方法可以为我们放回属于太阳系的行星的名字，喂养我们的蚯蚓，或者从相册中返回一张照片。下面只有长的那个标记是可以工作的方法：

```
$sun.getPlanet( ["Earth", "Mars", "Neptune"] )
## 不能将参数列表传递给$sun.Planets

$sisyphus.pushRock()
## Velocity 假定我意思是$sisyphus.getRock()

$book.setTitle( "Homage to Catalonia" )
## 不能传递一个参数列表
```

8. 形式引用符Formal Reference Notation

引用的简略符号如上所述，但是另外还有一种引用的形式符号，示例如下：

```
${mudSlinger}
${customer.Address}
${purchase.getTotal()}
```

在大多数情况下，我们将使用引用的简略符号，但在一些情况下，也需要拥戴哦形式引用符以便正确处理。

假定你正在纸片上构件一个句子，将使用$vice作为句子中名词的词根。我们的目标是允许人们选择词根，然后产生以下两种结果之一：

```
"Jack is a pyromaniac."
```

或者

```
"Jack is a kleptomaniac."。
```

在这种情况下，使用简略符号是不太充分的。考虑到下面的例子：

```
Jack is a $vicemaniac.
```

这里有个不确定性， Velocity 假定 $vicemaniac，（而不是 $vice）是一个你想要使用的标识符。找不到$vicemaniac的值，他将返回$vicemaniac。使用形式符号便可解决这个问题：

```
Jack is a ${vice}maniac
```

现在Velocity 知道$vice（而不是$vicemaniac）是一个引用。形式符号常用在饮用咋模板中和文本直接邻近的地方。

9. 安静引用符Quiet Reference Notation

当 Velocity 遇到一个位定义的引用时，其通常行为是输出这个引用的映像。比如，假设下面的引用出现在模板中的一部分：

```
<input type="text" name="email" value="$email"/>
```

当表单初次装入时，变量引用$email无值，你宁愿是一个空白域而不是具有值"$email"。使用安静引用符可以绕过Velocity的常规行为，在VTL中不用$email而是用$!email 符号。所以，上面的例子将会看起来像下面的样子：

```
<input type="text" name="email" value="$!email"/>
```

现在，当表单初次装入时， $email 仍然没有值，但是将输出空字符串而不是"$email"。

形式和安静引用符可以一起使用，如下所示：

```
<input type="text" name="email" value="$!{email}"/>
```

11. Case Substitution

现在你大致了解了引用，可以在模板中使用它们了。Velocity 采用了很多JAVA原理的优点，模板设计人员会发现非常容易使用。例如：

```
$foo

$foo.getBar()
## is the same as
$foo.Bar

$data.getUser("jon")
## is the same as
$data.User("jon")

$data.getRequest().getServerName()
## is the same as
$data.Request.ServerName
## is the same as
${data.Request.ServerName}
```

这个例子显示了引用的一些其他用法。Velocity 借鉴了Java的自省和组件bean特征，来解决引用名在上下文中作为对象和对象方法的问题。可以在你的模板的任何地方插入引用和求值。

Velocity, 建模在Sun Microsystems定义的BEAN规范之上，是大小写敏感的；开发者努力捕捉和纠正可能出现的用户错误。当方法getFoo() 在模板中通过$bar.foo引用时，Velocity 首先尝试$getfoo。如果失败，他会再尝试 $getFoo。类似地，当一个模板引用到 $bar.Foo， Velocity 将尝试 $getFoo() 先，然后尝试 getfoo()。

注意：模板中引用示例变量的问题仍然没有解决。只有引用等价于JavaBean的getter/setter 方法解决了。(比如 $foo.Name 解决了到类 Foo的 getName() 示例方法的引用，但不能引用Foo的一个公共实例变量Name)。

12. 指令

因为指令（使用脚本来有效操控JAVA代码的输出）允许页面设计员眞正专注于咱点的外观和内容设计，引用允许模板设计员为Web页面产生动态内容。

12.1. #set

`#set` 指令用来为引用设置相应的值。值可以被值派给变量引用或者是属性引用，而且赋值要在括号里括起来。

```
#set( $primate = "monkey" )
#set( $customer.Behavior = $primate )
```

赋值的左边必须是一个变量应用或者是属性引用。右边可以是下面的类型之一：

- 变量引用
- 字面字符串
- 属性引用
- 方法引用
- 字面数字
- 数组列表

这些例子演示了上述的每种类型：

```
#set( $monkey = $bill ) ## variable reference
#set( $monkey.Friend = "monica" ) ## string literal
#set( $monkey.Blame = $whitehouse.Leak ) ## property reference
#set( $monkey.Plan = $spindoctor.weave($web) ) ## method referen
ce
#set( $monkey.Number = 123 ) ##number literal
#set( $monkey.Say = ["Not", $my, "fault"] ) ## ArrayList
```

注意：最后一个例子中，在方括号[..] 中定义的项目可以被ArrayList 类定义的方法访问。比如，你可以使用$monkey.Say.get(0)访问上述的第一个元素。

右边也可以是一个简单的算术表达式:

```
#set( $value = $foo + 1 )
#set( $value = $bar - 1 )
#set( $value = $foo * $bar )
#set( $value = $foo / $bar )
```

如果右边是一个属性或方法引用，取值是NULL，他将不会赋值给左边。通过这种机制将一个存在的引用从上下文中删除是不可能的。这对Velocity的新手可能会混淆。例如：

```
#set( $result = $query.criteria("name") )
The result of the first query is $result

#set( $result = $query.criteria("address") )
The result of the second query is $result
```

如果， $query.criteria("name") 放回字符串"bill"，而$query.criteria("address") 返回null，上述VTL 将解释为：

```
The result of the first query is bill

The result of the second query is bill
```

这往往会给那些想构建#foreach循环来试图通过属性和方法引用来设置一个引用的新手带来困惑，下面马上通过#if指令测试一下。例如：

```
#set( $criteria = ["name", "address"] )

#foreach( $criterion in $criteria )

    #set( $result = $query.criteria($criterion) )

    #if( $result )
        Query was successful
    #end

#end
```

在上面的例子中，依靠$result 的去值来决定查询是否成功恐怕不是英明的做法。当$result 被#set设置后(添加到上下文中)，他就不能再被设值为null (从上下文中删除)。

我们对此的解决方法是预设$result 为 false。然后如果 $query.criteria() 调用失败，你就可以检查之。

```
#set( $criteria = ["name", "address"] )

#foreach( $criterion in $criteria )

    #set( $result = false )
    #set( $result = $query.criteria($criterion) )

    #if( $result )
        Query was successful
    #end

#end
```

不象其他Velocity 指令， #set 指令没有#end 语句。

12.2. 字面字符串

当使用#set 指令时，括在双引号中的字面字符串将解析和重新解释，如下所示：

```
#set( $directoryRoot = "www" )
#set( $templateName = "index.vm" )
#set( $template = "$directoryRoot/$templateName" )
$template
```

输出将会是：

```
www/index.vm
```

然而，当字面字符串括在单引号中时，他将不被解析：

```
#set( $foo = "bar" )
$foo
#set( $blargh = '$foo' )
$blargh
```

输出是：

```
Bar
 $foo
```

默认情况下，使用单引号来渲染未解析文本在Velocity是有效的。这种特征可以通过编辑velocity.properties 中的stringliterals.interpolate=false来改变。

12.3. 条件

12.3.1 If / ElseIf / Else

Velocity中的#if 指令允许在页面生成时，在IF条件为眞的情况下包含文本。例如：

```
#if( $foo )
   <strong>Velocity!</strong>
#end
```

变量 $foo 先求值，以决定是否为眞。在这两种情况下为眞： (i) $foo 是一个逻辑变量并具有眞的值，或者 (ii) 值非空。要记住Velocity 上下文仅包括对象，所以当我们说“布尔”'boolean'时，他会被表示为“布尔类”(Boolean class)。这对即使是返回布尔类型的方法也是眞的—自省架构将返回一个具有相同逻辑值的布尔类。

如果求值为眞时， #if 和 #end 语句之间的内容将输出。在这种情况下，如果 $foo 为眞，输出将是"Velocity!"。相反，如果 $foo 具有一个null 值，或者逻辑假，语句求值为假，则没有输出。

一个 #elseif 或者 #else 项可以用在#if 语句中。请注意， Velocity 模板引擎将在第一个为眞的表达式时停止。下面的例子中，假设$foo 具有值15 而 $bar 等于 6。

```
#if( $foo < 10 )
    <strong>Go North</strong>
#elseif( $foo == 10 )
    <strong>Go East</strong>
#elseif( $bar == 6 )
    <strong>Go South</strong>
#else
    <strong>Go West</strong>
#end
```

在这个例子中，$foo 大于10，所以前面两个比较失败。接下来比较$bar 和6，结果为眞，所以输出为Go South。

请注意在现在， Velocity的数值比较约束为整型—其他类型都将求值为false。仅有一个例外是等于'=='，这时Velocity 要求等号两边的对象具有相同的类型。

12.3.2 关系和逻辑操作符

Velocity 使用等式操作符来决定两个变量间的关系。这里是一个简单的例子演示如何使用等式操作符：

```
#set ($foo = "deoxyribonucleic acid")
#set ($bar = "ribonucleic acid")

#if ($foo == $bar)
 In this case it's clear they aren't equivalent. So...
#else
 They are not equivalent and this will be the output.
#end
```

Velocity 也具有逻辑AND, or 和 NOT 操作符。更进一步的信息，请看VTL参考手册 VTL Reference Guide 。下面是一些演示如何使用逻辑操作符的例子：

```
## logical AND

#if( $foo && $bar )
   <strong> This AND that</strong>
#end
```

例子中#if() 指令仅在$foo 和$bar 斗为真的时候才为真。如果$foo 为假，则表达式也为假；并且 $bar 将不被求值。如果 $foo 为真，Velocity 模板引擎将继续检查$bar;的值，如果 $bar 为真，则整个表达式为真。并且输出This AND that 。如果 $bar 为假，将没有输出因为整个表达式为假。

逻辑OR 的工作方式相同，唯一的例外是其中一个表达式要被求值，以便决定整个表达式是否为真。请看下面的例子：

```
## logical or

#if( $foo || $bar )
    <strong>This or That</strong>
#end
```

如果 $foo 为真，Velocity 模板引擎就不需要去察看$bar 的值，不管 $bar 是否为真，真个表达式都为真，因此输出This or That 。如果 $foo 为假，$bar 就必须检查其值了。在这种情况下，如果$bar 也是为假，表达式将为假，没有任何输出。当然，如果$bar 为真，则真个表达式为真，输出This or That。

对于逻辑NOT 操作符，只有一个操作数：

```
##logical NOT

#if( !$foo )
 <strong>NOT that</strong>
#end
```

这里，如果$foo 为真，!$foo 求值为假，没有输出。如果$foo 为假，!$foo 求值为真，输出NOT that 。请当心，不要和安静引用quiet reference $!foo 混淆它们是完全不同的。

12.4. 循环

12.4.1. Foreach 循环

#foreach 元素允许进行循环，例如：

```
<ul>
#foreach( $product in $allProducts )
    <li>$product</li>
#end
</ul>
```

这个#foreach 循环将导致$allProducts 列表 (对象) 为查询所有的产品$products (目标)遍历一遍。每次经过循环，从$allProducts取得的值将置于$product 变量之中。

$allProducts 变量的内容是一个矢量，一个哈希表或者数组。赋给$product 变量的值是一个Java 对象并且可以从一个类似的变量引用。例如，如果 $product 真是一个Java的产品类，其名称可以通过引用$product.Name 方法来检索(即: $Product.getName())。

我们假定 $allProducts 是一个哈希表。如果你想检索关键字的值或者在哈希表中的对象，你可以使用以下的代码：

```
<ul>
#foreach( $key in $allProducts.keySet() )
    <li>Key: $key -> Value: $allProducts.get($key)</li>
#end
</ul>
```

Velocity 提供一个更容易的方式或的循环计数，以便你可以做下面类似的工作：

```
<table>
#foreach( $customer in $customerList )
    <tr><td>$velocityCount</td><td>$customer.Name</td></tr>
#end
</table>
```

循环计数变量的缺省名称是$velocityCount，在velocity.properties 配置文件中标明。默认情况下，该变量从1开始计数，但是可以在velocity.properties 文件中设为从0或者1开始。下面是velocity.properties 文件中循环变量设置一节:

```
# Default name of the loop counter
# variable reference.
directive.foreach.counter.name = velocityCount

# Default starting value of the loop
# counter variable reference.
directive.foreach.counter.initial.value = 1
```

12.5. 包含

#include 脚本元素允许模板设计人员包含（导入）本地文件，这个文件将插入到 #include 指令被定义的地方。文件的内容并不通过模板引擎来渲染。处于安全的原因，被包含的文件只可以放在TEMPLATE_ROOT下。

```
#include( "one.txt" )
```

#include 指令引用的文件在双引号内。如果超过一个文件，其间用逗号隔开。

```
#include( "one.gif","two.txt","three.htm" )
```

被包含的文件并不是一定要用文件名来引用，事实上，最好的办法是使用变量而不是文件名。这在根据规则决定何时提交页面时，决定目标输出是很有用的。

```
#include( "greetings.txt", $seasonalstock )
```

12.6. 解析

#parse 脚本元素允许页面设计员导入包含VTL的本地文件。 Velocity将解析和渲染指定的模板。

```
#parse( "me.vm" )
```

就象 #include 指令，#parse 可以使用变量而不是一个实在的模板文件。#parse 引用的模板文件必须包含的TEMPLATE_ROOT指定的目录之下。和 #include 指令不一样， #parse 只有一个参数。

VTL 模板templates can have #parse statements referring to templates that in turn have #parse statements. By default set to 10, the parse_directive.maxdepth line of the velocity.properties allows users to customize maximum number of #parse referrals that can occur from a single template. (Note: If the parse_directive.maxdepth property is absent from the velocity.properties file, Velocity will set this default to 10.) Recursion is permitted, for example, if the template dofoo.vm contains the following lines:

```
Count down.
#set( $count = 8 )
#parse( "parsefoo.vm" )
All done with dofoo.vm!
```

It would reference the template parsefoo.vm, which might contain the following VTL:

```
$count
#set( $count = $count - 1 )
#if( $count > 0 )
    #parse( "parsefoo.vm" )
#else
    All done with parsefoo.vm!
#end
```

After "Count down." is displayed, Velocity passes through parsefoo.vm, counting down from 8. When the count reaches 0, it will display the "All done with parsefoo.vm!" message. At this point, Velocity will return to dofoo.vm and output the "All done with dofoo.vm!" message.

12.7. 停止

#stop 脚本允许模板设计员停止模板引擎的执行，并返回。这通常用作调试。

```
#stop
```

12.10. 宏

#macro 脚本元素允许模板设计者在VTL 模板中定义重复的段。 Velocimacros 不管是在复杂还是简单的场合都非常有用。下面这个Velocimacro，仅用来节省击键和减少排版错误，介绍了一些Velocity宏的概念。

```
#macro( d )
<tr><td></td></tr>
#end
```

在例子中，Velocimacro定义为d，它可以象调用其他VTL指令一样的形式来进行调用：

```
#d()
```

当这个模板被调用时， Velocity 将 #d() 替换为一个单行的空表格。

Velocimacro 可以带一些参数，也可以不带参数（如上例所示）。但在他被调用时，所带的参数必须和其定义时的参数一样。很多Velocimacros 定义为不止一个参数。下面这个宏带有两个参数，一个颜色，一个数组。

```
#macro( tablerows $color $somelist )
#foreach( $something in $somelist )
    <tr><td bgcolor=$color>$something</td></tr>
#end
#end
```

在这个例子中定义的Velocimacro，名为tablerows, 要求两个参数。第一个参数代替$color, 第二个代替$somelist。

可以写进VTL 模板中的东西都可以写进Velocimacro 的主体部分。tablerows 宏其实是一个foreach 语句。在#tablerows 宏的定义中有两个#ende语句，第一个属于#foreach, 第二个结束宏定义。

```
#set( $greatlakes = ["Superior","Michigan","Huron","Erie","Ontar
io"] )
#set( $color = "blue" )
<table>
    #tablerows( $color $greatlakes )
</table>
```

请注意$greatlakes 替换了$somelist。这样，当#tablerows 宏被调用时，将产生以下输出：

```
<table>
    <tr><td bgcolor="blue">Superior</td></tr>
    <tr><td bgcolor="blue">Michigan</td></tr>
    <tr><td bgcolor="blue">Huron</td></tr>
    <tr><td bgcolor="blue">Erie</td></tr>
    <tr><td bgcolor="blue">Ontario</td></tr>
</table>
```

Velocimacros 在Velocity 模板语句内定义，这意味着它在同一站点内的其他Velocity模板中并不有效。定义一个宏，并使其与其他模板共享很具有明显的优点：他减少了在大量的模板内重复定义宏的工作，并减少了出错的机会，并确保对其他宏的改变对其他所有模板有效。

但如果 #tablerows($color $list) 宏是在一个Velocimacros 模板库内定义的，它就可以被其他常规模板所用。当然，它可以用于各种目的，也可重用多次。在表示所有真菌类（fungi）的mushroom.vm 模板中，#tablerows 宏可以被用来列出典型的蘑菇。

```
#set( $parts = ["volva","stipe","annulus","gills","pileus"] )
#set( $cellbgcol = "#CC00FF" )
<table>
#tablerows( $cellbgcol $parts )
</table>
```

我们对mushroom.vm执行请求，Velocity 将在模板库内找到#tablerows 宏 (在velocity.properties 文件中定义)并产生以下输出：

```
<table>
    <tr><td bgcolor="#CC00FF">volva</td></tr>
    <tr><td bgcolor="#CC00FF">stipe</td></tr>
    <tr><td bgcolor="#CC00FF">annulus</td></tr>
    <tr><td bgcolor="#CC00FF">gills</td></tr>
    <tr><td bgcolor="#CC00FF">pileus</td></tr>
</table>
```

12.10.1. Velocimacro 参数

Velocimacros 的参数可以是以下的VTL元素：

- 引用（Reference）: 以 '$' 打头的元素
- 字面字符串（String literal）: 比如"$foo" 或 'hello'
- 字面数字: 1, 2 ....
- 整数范围: [1..2] 或 [$foo .. $bar]
- 对象数组: ["a", "b", "c"]
- 布尔真
- 布尔假

当把引用作为参数传递给Velocimacros时，请注意引用是按"名字"传递的。这意味着他们的值在每次使用他们的Velocimacro中产生。这个特性允许你在方法调用是传递引用，并在每次使用时进行方法调用。例如，Fo，当调用下面的Velocimacro时，

```
#macro( callme $a )
         $a $a $a
     #end

     #callme( $foo.bar() )
```

结果是，在方法bar() 中，引用 $foo 被调用了3次。

咋看时，这个特征让人吃惊，当当你考虑一下Velocimacros的原本动机 – 在VTL模板中避免很多"剪切复制"操作—你就会明白。它允许你将无状态对象，比如在一个颜色表格行内重复产生一些颜色次序的对象，传递给Velocimacro。

如果你需要使用这个特征，你通常可以从方法内取得一个值，作为一个新的引用传递给宏：

```
 #set( $myval = $foo.bar() )
     #callme( $myval )
```

12.10.2. Velocimacro 属性

在velocity.properties 文件中有数行定义可以用来灵活实现Velocimacros。详细情况请参见开发指南（Developer Guide）。

velocimacro.library –是一个逗号分隔的所有Velocimacro 模板库的列表。默认情况下， Velocity 搜寻一个单一的库VM_global_library.vm.。预先配置的模板路径用来查找Velocimacro 库。

velocimacro.permissions.allow.inline –这个属性决定Velocimacros 是否可以在常规模板内定义，取值为逻辑True或者False。默认情况下，设置为true，允许设计者在产规模板内定义宏。

velocimacro.permissions.allow.inline.to.replace.global –逻辑true 或者false，允许标明是否允许在常规模板内定义的Velocimacro代替在模板库中定义并通过velocimacro.library属性在启动时装入的全局宏。默认设置为false。

velocimacro.permissions.allow.inline.local.scope –逻辑true 或者false，默认值为false。控制是否在模板内定义的Velocimacros仅在定义它的模板内可见。换句话说，如果设置为true，一个模板可以定义仅能被他所用的宏。你可以用它来做一些漂亮的宏，如果一个全局调用另一个全局宏，在局部（inline）范围内，当被一个模板调用时，该模板可以定义一个被第一个全局宏调用的第二个全局宏的私有实现。其他所有模板都不受影响。

velocimacro.context.localscope –逻辑值true 或者 false，缺省值为false。但设置为true时，所有在Velocimacro 内通过 #set() 进行的修改都将被视为Velocimacro 的本地行为，不会影响到其上下文。

velocimacro.library.autoreload –此属性控制Velocimacro 库的自动载入。缺省值为false。如果设置为true，被调用的Velocimacro得源库将被检查是否改变，并在必要是重新载入。这将使你可以改变和测试Velocimacro 库，而不必重新启动应用服务器或者servlet容器，就象你工作在常规模板一样。这个模时仅在资源载入器的缓存模时被关闭的情况下有效 (如file.resource.loader.cache = false)。此特征为开发时设计，不要在生产模式时使用。

12.10.3. Velocimacro Trivia

当前， Velocimacros 在其首次在模版中使用前必须首先定义它。这意味着，#macro() 宣称应该在使用Velocimacros之前。

如果你想#parse() 一个包含#macro() 指令的模板，记住这个非常重要。因为#parse() 在运行时发生，解析器在解析时要决定是否模版中一个看起来像VM的元素眞是VM，所以解析一系列VM 宣称可能并不能如愿地工作的很好。为避免如此，可以简单地使用velocimacro.library 的办法，使Velocity 在启动时载入VM。

13. Getting literal

Velocity使用特殊字符$和#来帮助它工作，所以如果要在template里使用这些特殊字符要格外小心。本节将讨论$字符。

13.1. 货币字符

这是没有问题的："I bought a 4 lb. sack of potatoes at the farmer's market for only $2.50!"，VTL中使用$2.5这样的货币标识是没有问题得的，VTL不会将它错认为是一个reference，因为VTL中的reference总是以一个大写或者小写的字母开始。

13.2. 转义 有效的 VTL 指令

某些情况使用Velocity可能会觉得很烦恼。逃避特殊符是处理出现在你的模板中VTL特殊符有效方法，就是使用反斜杠(“\”)。

```
#set( $email = "foo" )
$email
```

假如Velocity在你的模板中遇到$email，它会搜索上下文，得到相应的值。这里的输出是foo，因为$email被定义了。假如$email没有被定义，输出会是$email。

设想$email被定义了(例如，它的值是foo)，而且你想输出$email。这里有几种方法能达到目的，但是最简单的是使用逃避符。

```
## The following line defines $email in this template:
#set( $email = "foo" )
$email
\$email
```

将显示为：

```
foo
$email
```

注意到“\”屏蔽了左边的“$”。屏蔽左边规则，使得\$email显示为\$email。那些例子与$email没有定义相比较。

```
$email
\$email
\\$email
\\\$email
```

将显示为：

```
$email
\$email
\\$email
\\\$email

\\$email\\\$email
```

注意Velocity处理定义了的references与没有定义的不一样。这里set$foo的值为gibbous。

```
#set( $foo = "gibbous" )
$moon = $foo
```

输出会是： `$moon=gibbous,$moon` 按照字面上输出因为它没有定义，gibbous替代$foo输出。避开VTL的directives还有其他方法，在Directives那章节会更详细描述。

13.3. 转义 无效的 VTL 指令

VTL 可以通过反斜杠("/")来进行转义，directives can be escaped with the backslash character in a manner similar to valid VTL references.

```
## #include( "a.txt" ) renders as <contents of a.txt>
#include( "a.txt" )

## /#include( "a.txt" ) renders as /#include( "a.txt" )
/#include( "a.txt" )

## //#include ( "a.txt" ) renders as /<contents of a.txt>
//#include ( "a.txt" )
```

在转义在一个单一指令内包含多个脚本元素（比如f-else-end语句）的指令时应多加小心。下面是一个典型的VTL if语句；

```
#if( $jazz )
    Vyacheslav Ganelin
#end
```

如果 $jazz为 true，输出是

```
Vyacheslav Ganelin
```

如果 $jazz 为false，将没有输出。转义脚本元素将改变输出。考虑下面的情况；

```
/#if( $jazz )
    Vyacheslav Ganelin
/#end
```

不管 $jazz 是真或假，输出都是

```
#if($ jazz )
     Vyacheslav Ganelin
 #end
```

事实上，因为所有脚本元素都被转义了， $jazz 永远不会被求值。将设反斜杠在被合法转义的脚本元素之前

```
//#if( $jazz )
   Vyacheslav Ganelin
//#end
```

这时，如果$jazz 为眞，输出是

```
/ Vyacheslav Ganelin
/
```

为理解这个情况，请注意在一个新行结束是将在输出中忽略新的一行。因此，经过#if()前的'//' 加工后，#if()块紧跟第一个'/'。最后一个/位于新的一行，因为在'Ganelin'后又一个新行，所以，最后的那个位于#end 之前的//是语句块的一部分。

如果 $jazz 为false，这里将没有输出。注意，在开始破坏了if语句的情况将不能被正确转义：

```
///#if( $jazz )
    Vyacheslave Ganelin
//#end
```

这里，#if 被转义，但有一个#end 被保留了；所以有多个结束语句将导致解析错误。

14. VTL 格式化问题

虽然在本指南中的VTL经常显示在新行中或者有空格，但是下面的VTL

```
#set( $imperial = ["Munetaka","Koreyasu","Hisakira","Morikune"]
)
#foreach( $shogun in $imperial )
    $shogun
#end
```

和下面的写法同样有效。

```
Send me #set($foo = ["$10 and ","a cake"])#foreach($a in $foo)$a
 #end please.
```

Velocity的行为并不受空格的影响，前述的指令也可以写成：

```
Send me
#set( $foo = ["$10 and ","a cake"] )
#foreach( $a in $foo )
$a
#end
please.
```

或者

```
Send me
#set($foo       = ["$10 and ","a cake"])
                 #foreach           ($a in $foo )$a
         #end please.
```

上面每种写法结果都一样。

15. 其它特征和杂项

15.1. 数学特征

Velocity 有一些内建的数学功能，可以使用set指令用在模版中。下面的共识分别演示了加减乘除运算：

```
#set( $foo = $bar + 3 )
#set( $foo = $bar - 4 )
#set( $foo = $bar * 6 )
#set( $foo = $bar / 2 )
```

当进行除法运算时，结果将会是整数。When a division operation is performed, the result will be an integer. 余数则可以通过模(%)运算获得。

```
#set( $foo = $bar % 5 )
```

在Velocity 中，只有整数可以进行数学运算；如果执行非整数的数学运算，将被记录下来，并返回null 。

15.2. 范围操作符

范围操作符可以和#set 和#foreach 语句一起使用。有助于产生一个整数的目标数组，范围操作符有以下的结构：

```
[n..m]
```

n 和 m 都必须是整数或者可以产生整数。不管 m 大于或者小于n 都没关系；在m小于n这种情况下，范围可以向下计数。下面是使用范围操作符的例子：

第一个例子

```
#foreach( $foo in [1..5] )
$foo
#end
```

第二个例子

```
#foreach( $bar in [2..-2] )
$bar
#end
```

第三个例子

```
#set( $arr = [0..1] )
#foreach( $i in $arr )
$i
#end
```

第四个例子

```
[1..3]
```

他们分别产生一下输出

```
1 2 3 4 5

2 1 0 -1 -2

0 1

 [1..3]
```

范围操作符和#set 和#foreach 指令一起使用时，只是产生数组。

页面设计人员在设计具有相同尺寸的表格时，有时没有足够的数据来填充，他们会发现范围操作符非常有用。

15.3. 进阶：转义和!

当一个引用被! 字符处于静寂模式，并且! 字符在转义符/ 前出现，应用将用一种特别的方式处理。请注意意他和常规转义的不同，下面这种情况/ 先于! 出现:

```
#set( $foo = "bar" )
$/!foo
$/!{foo}
$//!foo
$///!foo
```

这样将被加工成

```
$!foo
$!{foo}
$/!foo
$//!foo
```

对比常规转义，/先于 $:

```
/$foo
/$!foo
/$!{foo}
//$!{foo}
```

这是结果是：

```
/$foo
/$!foo
/$!{foo}
/bar
```

15.4. Velocimacro 杂记

本节是关于Velocimacros的一个小型FAQ。本届内容会不时更新，所以请常来检查新的内容，

注：本节中，'Velocimacro' 将简写为'VM'。

Q:是否可以使用指令directive 或者 VM 作为另一个VM的参数? 例如: #center(#bold("hello"))

A: 不行。指令不能用作指令的参数，而大多数情况下，作为实际的应用，VM就是指令。

不过也有一些办法。一个简单的做法是使用双引号来加工你的内容。所以，你可以这样：

```
#set($stuff = "#bold('hello')" )
#center( $stuff )
```

甚至可以节省一个步骤：

```
#center( "#bold( 'hello' )" )
```

请注意，后面这个例子中，参数是在VM内部被求值，不是在调用的那一层次上。换句话说，被传入的VM的参数是整个被传入的，并且在传入的VM内部被求值。所以我们可以这样做：

```
#macro( inner $foo )
 inner : $foo
#end

#macro( outer $foo )
   #set($bar = "outerlala")
   outer : $foo
#end

#set($bar = 'calltimelala')
#outer( "#inner($bar)" )
```

这里，输入将会是：

```
Outer : inner : outerlala
```

因为"#inner($bar)" 的求值发生在#outer()内部，所以在#outer() 内设置的$bar得值会是其使用的值。

这是一个有意的保护特征—参数按名称传递给VM，所以可以将象状态引用的东西传给VM ，比如：

```
#macro( foo $color )
 <tr bgcolor=$color><td>Hi</td></tr>
 <tr bgcolor=$color><td>There</td></tr>
#end

#foo( $bar.rowColor() )
```

rowColor() 被重复调用而不是一次。为避免如此，可以调用VM外部的方法，然后将值传递给VM.

```
#set($color = $bar.rowColor())
#foo( $color )
```

Q:是否可以通过#parse()注册VM？

A：当前， Velocimacros 在其首次在模版中使用前必须首先定义它。这意味着，#macro() 宣称应该在使用Velocimacros之前。

如果你想#parse() 一个包含#macro() 指令的模板，记住这个非常重要。因为#parse() 在运行时发生，解析器在解析时要决定是否模版中一个看起来像VM的元素眞是VM，所以解析一系列VM 宣称可能并不能如愿地工作的很好。为避免如此，可以简单地使用velocimacro.library 的办法，使Velocity 在启动时载入VM。

Q. 什么是VM自动载入（Velocimacro Autoreloading）？

A. 这是一个属性，在开发时使用，而不时运行时：

```
 velocimacro.library.autoreload
```

默认值为false。当设置为true时，连同<type>.resource.loader.cache 属性设置为false（这里 是使用的资源载入器的名称，比如'file'），Velocity 引擎在你创建VM库文件是将自动载入其改变，这样你就不必将其导入servlet 引擎（或者应用程序）中，或者用其他手段来使其自动重新载入。

下面是一个简单的设置配置组合：

```
file.resource.loader.path = templates
    file.resource.loader.cache = false
    velocimacro.library.autoreload = true
```

注意在生产状态（运行时）不要使其打开。

15.5. 字符串联

开发者常问的一个问题是“我如何进行字符串串联？”是否有类似于JAVA中的'+'操作符？

为了串联VTL中的引用，你不得不将它们“放在一起”。而你想要放置在一起的上下文很重要，下面举例说明。

在常规“笨办法”模板中：

```
 #set( $size = "Big" )
 #set( $name = "Ben" )

The clock is $size$name.
```

输出将会是：'The clock is BigBen’。我们来看更有趣的事情，比如，当你想串联一个字符串并传递给一个方法，或者设置一个新的引用，可以这样：

```
#set( $size = "Big" )
#set( $name = "Ben" )

#set($clock = "$size$name" )

The clock is $clock.
```

结果是一样的。作为最后一个例子，当你想混合“静态”字符串到引用中，你可能需要使用“形式引用”：

```
#set( $size = "Big" )
#set( $name = "Ben" )
#set($clock = "${size}Tall$name" )
```

现在，输出将会是'The clock is BigTallBen'。

Google Guava官方教程（中文版）

原文链接 译文链接 译者: 沈义扬，罗立树，何一昕，**武祖 校对**：方腾飞

引言

Guava工程包含了若干被Google的 Java项目广泛依赖 的核心库，例如：集合 [collections] 、缓存 [caching] 、原生类型支持 [primitives support] 、并发库 [concurrency libraries] 、通用注解 [common annotations] 、字符串处理 [string processing] 、I/O 等等。 所有这些工具每天都在被Google的工程师应用在产品服务中。

查阅Javadoc并不一定是学习这些库最有效的方式。在此，我们希望通过此文档为Guava中最流行和最强大的功能，提供更具可读性和解释性的说明。

译文格式说明

- Guava中的类被首次引用时，都会链接到Guava的API文档。
  如：Optional<T>。
- Guava和JDK中的方法被引用时，一般都会链接到Guava或JDK的API文档，一些人所共知的JDK方法除外。
  如： `[Optional.of(T)](http://docs.guava-libraries.googlecode.com/`
  , Map.get(key)。
- 译者对文档的额外说明以斜体显示，并且以“译者注__：”开始。

目录

1. 基本工具 [Basic utilities]

让使用Java语言变得更舒适

1.1 使用和避免null：null是模棱两可的，会引起令人困惑的错误，有些时候它让人很不舒服。很多Guava工具类用快速失败拒绝null值，而不是盲目地接受

1.2 前置条件: 让方法中的条件检查更简单

1.3 常见Object方法: 简化Object方法实现，如hashCode()和toString()

1.4 排序: Guava强大的”流畅风格比较器”

1.5 Throwables：简化了异常和错误的传播与检查

2. 集合[Collections]

Guava对JDK集合的扩展，这是Guava最成熟和为人所知的部分

2.1 不可变集合: 用不变的集合进行防御性编程和性能提升。

2.2 新集合类型: multisets, multimaps, tables, bidirectional maps等

2.3 强大的集合工具类: 提供java.util.Collections中没有的集合工具

2.4 扩展工具类：让实现和扩展集合类变得更容易，比如创建 `Collection` 的装饰器，或实现迭代器

3. 缓存[Caches]

Guava Cache：本地缓存实现，支持多种缓存过期策略

4. 函数式风格[Functional idioms]

Guava的函数式支持可以显著简化代码，但请谨慎使用它

5. 并发[Concurrency]

强大而简单的抽象，让编写正确的并发代码更简单

5.1 ListenableFuture：完成后触发回调的Future

5.2 Service框架：抽象可开启和关闭的服务，帮助你维护服务的状态逻辑

6. 字符串处理[Strings]

非常有用的字符串工具，包括分割、连接、填充等操作

7. 原生类型[Primitives]

扩展 JDK 未提供的原生类型（如int、char）操作， 包括某些类型的无符号形式

8. 区间[Ranges]

可比较类型的区间API，包括连续和离散类型

9. I/O

简化I/O尤其是I/O流和文件的操作，针对Java5和6版本

10. 散列[Hash]

提供比 `Object.hashCode()` 更复杂的散列实现，并提供布鲁姆过滤器的实现

11. 事件总线[EventBus]

发布-订阅模式的组件通信，但组件不需要显式地注册到其他组件中

12. 数学运算[Math]

优化的、充分测试的数学工具类

13. 反射[Reflection]

Guava 的 Java 反射机制工具类

1-基本工具

1.1-使用和避免null

原文链接 译文链接 译者: 沈义扬

Doug Lea 说，“*Null* 真糟糕。”

当*Sir C. A. R. Hoare* 使用了*null*引用后说，”使用它导致了十亿美金的错误。”

轻率地使用null可能会导致很多令人惊愕的问题。通过学习Google底层代码库，我们发现95%的集合类不接受null值作为元素。我们认为， 相比默默地接受null，使用快速失败操作拒绝null值对开发者更有帮助。

此外，Null的含糊语义让人很不舒服。Null很少可以明确地表示某种语义，例如，Map.get(key)返回Null时，可能表示map中的值是null，亦或map中没有key对应的值。Null可以表示失败、成功或几乎任何情况。使用Null以外的特定值，会让你的逻辑描述变得更清晰。

Null确实也有合适和正确的使用场景，如在性能和速度方面Null是廉价的，而且在对象数组中，出现Null也是无法避免的。但相对于底层库来说，在应用级别的代码中，Null往往是导致混乱，疑难问题和模糊语义的元凶，就如同我们举过的Map.get(key)的例子。最关键的是，Null本身没有定义它表达的意思。

鉴于这些原因，很多Guava工具类对Null值都采用快速失败操作，除非工具类本身提供了针对Null值的因变措施。此外，Guava还提供了很多工具类，让你更方便地用特定值替换Null值。

具体案例

不要在Set中使用null，或者把null作为map的键值。使用特殊值代表null会让查找操作的语义更清晰。

如果你想把null作为map中某条目的值，更好的办法是 不把这一条目放到map中，而是单独维护一个”值为null的键集合” (null keys)。Map 中对应某个键的值是null，和map中没有对应某个键的值，是非常容易混淆的两种情况。因此，最好把值为null的键分离开，并且仔细想想，null值的键在你的项目中到底表达了什么语义。

如果你需要在列表中使用null——并且这个列表的数据是稀疏的，使用Map<Integer, E>可能会更高效，并且更准确地符合你的潜在需求。

此外，考虑一下使用自然的null对象——特殊值。举例来说，为某个enum类型增加特殊的枚举值表示null，比如java.math.RoundingMode就定义了一个枚举值UNNECESSARY，它表示一种不做任何舍入操作的模式，用这种模式做舍入操作会直接抛出异常。

如果你真的需要使用null值，但是null值不能和Guava中的集合实现一起工作，你只能选择其他实现。比如，用JDK中的Collections.unmodifiableList替代Guava的ImmutableList

Optional

大多数情况下，开发人员使用null表明的是某种缺失情形：可能是已经有一个默认值，或没有值，或找不到值。例如，Map.get返回null就表示找不到给定键对应的值。

Guava用Optional<T>表示可能为null的T类型引用。一个Optional实例可能包含非null的引用（我们称之为引用存在），也可能什么也不包括（称之为引用缺失）。它从不说包含的是null值，而是用存在或缺失来表示。但Optional从不会包含null值引用。

```
Optional<Integer> possible = Optional.of(5);

possible.isPresent(); // returns true

possible.get(); // returns 5
```

Optional无意直接模拟其他编程环境中的”可选” or “可能”语义，但它们的确有相似之处。

Optional最常用的一些操作被罗列如下：

创建**Optional**实例（以下都是静态方法）：

`Optional.of(T)`	创建指定引用的**Optional**实例，若引用为**null**则快速失败
`Optional.absent()`	创建引用缺失的Optional实例
`Optional.fromNullable(T)`	创建指定引用的Optional实例，若引用为null则表示缺失

用**Optional**实例查询引用（以下都是非静态方法）：

`boolean isPresent()`	如果**Optional**包含非**null**的引用（引用存在），返回**true**
`T get()`	返回Optional所包含的引用，若引用缺失，则抛出java.lang.IllegalStateException
`T or(T)`	返回Optional所包含的引用，若引用缺失，返回指定的值
`T orNull()`	返回Optional所包含的引用，若引用缺失，返回null
Set<T> asSet()	返回Optional所包含引用的单例不可变集，如果引用存在，返回一个只有单一元素的集合，如果引用缺失，返回一个空集合。

使用**Optional**的意义在哪儿？

使用Optional除了赋予null语义，增加了可读性，最大的优点在于它是一种傻瓜式的防护。Optional迫使你积极思考引用缺失的情况，因为你必须显式地从Optional获取引用。直接使用null很容易让人忘掉某些情形，尽管FindBugs可以帮助查找null相关的问题，但是我们还是认为它并不能准确地定位问题根源。

如同输入参数，方法的返回值也可能是null。和其他人一样，你绝对很可能会忘记别人写的方法method(a,b)会返回一个null，就好像当你实现method(a,b)时，也很可能忘记输入参数a可以为null。将方法的返回类型指定为Optional，也可以迫使调用者思考返回的引用缺失的情形。

其他处理**null**的便利方法

当你需要用一个默认值来替换可能的null，请使用 `Objects.firstNonNull(T, T)` 方法。如果两个值都是null，该方法会抛出NullPointerException。Optional也是一个比较好的替代方案，例如：Optional.of(first).or(second).

还有其它一些方法专门处理null或空字符串：emptyToNull(String)，`nullToEmpty(String)` ，`isNullOrEmpty(String)` 。我们想要强调的是，这些方法主要用来与混淆null/空的API进行交互。当每次你写下混淆null/空的代码时，Guava团队都泪流满面。（好的做法是积极地把null和空区分开，以表示不同的含义，在代码中把null和空同等对待是一种令人不安的坏味道。

1.2-前置条件

原文链接 译文链接 译者: 沈义扬

前置条件：让方法调用的前置条件判断更简单。

Guava在Preconditions类中提供了若干前置条件判断的实用方法，我们强烈建议在Eclipse中静态导入这些方法。每个方法都有三个变种：

- 没有额外参数：抛出的异常中没有错误消息；
- 有一个Object对象作为额外参数：抛出的异常使用Object.toString() 作为错误消息；
- 有一个String对象作为额外参数，并且有一组任意数量的附加Object对象：这个变种处理异常消息的方式有点类似printf，但考虑GWT的兼容性和效率，只支持%s指示符。例如：

```
checkArgument(i >= 0, "Argument was %s but expected nonnegative"
, i);
checkArgument(i < j, "Expected i < j, but %s > %s", i, j);
```

方法声明（不包括额外参数）	描述
`checkArgument(boolean)`	检查boolean是为true，用来检传递给方法的数。
`checkNotNull(T)`	检查value是否null，该方法直返回value，因可以内嵌使用checkNotNull
`checkState(boolean)`	用来检查对象的某些状态。
`checkElementIndex(int index, int size)`	检查index作为引值对某个列表、字符串或数组是否有效。index>=0 && index<size *
`checkPositionIndex(int index, int size)`	检查index作为置值对某个列表、字符串或数组是否有效。index>=0 && index<=size *
`checkPositionIndexes(int start, int end, int size)`	检查[start, end示的位置范围对某个列表、字符串或数组是否有效*

译者注：

*索引值常用来查找列表、字符串或数组中的元素，如*List.get(int), String.charAt(int)*

*位置值和位置范围常用来截取列表、字符串或数组，如*List.subList(int，int), String.substring(int)*

相比Apache Commons提供的类似方法，我们把Guava中的Preconditions作为首选。Piotr Jagielski在他的博客中简要地列举了一些理由：

- 在静态导入后，Guava方法非常清楚明晰。checkNotNull清楚地描述做了什么，会抛出什么异常；
- checkNotNull直接返回检查的参数，让你可以在构造函数中保持字段的单行赋值风格：this.field = checkNotNull(field)
- 简单的、参数可变的printf风格异常信息。鉴于这个优点，在JDK7已经引

入Objects.requireNonNull的情况下，我们仍然建议你使用checkNotNull。

在编码时，如果某个值有多重的前置条件，我们建议你把它们放到不同的行，这样有助于在调试时定位。此外，把每个前置条件放到不同的行，也可以帮助你编写清晰和有用的错误消息。

1.3-常见Object方法

原文链接 译者: 沈义扬

equals

当一个对象中的字段可以为null时，实现Object.equals方法会很痛苦，因为不得不分别对它们进行null检查。使用 `Objects.equal` 帮助你执行null敏感的equals判断，从而避免抛出NullPointerException。例如:

```
Objects.equal("a", "a"); // returns true
Objects.equal(null, "a"); // returns false
Objects.equal("a", null); // returns false
Objects.equal(null, null); // returns true
```

注意：*JDK7*引入的*Objects*类提供了一样的方法 `_Objects.equals_` 。

hashCode

用对象的所有字段作散列[hash]运算应当更简单。Guava的 `Objects.hashCode(Object...)` 会对传入的字段序列计算出合理的、顺序敏感的散列值。你可以使用Objects.hashCode(field1, field2, ..., fieldn)来代替手动计算散列值。

注意：*JDK7*引入的*Objects*类提供了一样的方法 `_Objects.hash(Object...)_`

toString

好的toString方法在调试时是无价之宝，但是编写toString方法有时候却很痛苦。使用 Objects.toStringHelper可以轻松编写有用的toString方法。例如：

```
// Returns "ClassName{x=1}"
Objects.toStringHelper(this).add("x", 1).toString();
// Returns "MyObject{x=1}"
Objects.toStringHelper("MyObject").add("x", 1).toString();
```

compare/compareTo

实现一个比较器[Comparator]，或者直接实现Comparable接口有时也伤不起。考虑一下这种情况：

```
class Person implements Comparable<Person> {
  private String lastName;
  private String firstName;
  private int zipCode;

  public int compareTo(Person other) {
    int cmp = lastName.compareTo(other.lastName);
    if (cmp != 0) {
      return cmp;
    }
    cmp = firstName.compareTo(other.firstName);
    if (cmp != 0) {
      return cmp;
    }
    return Integer.compare(zipCode, other.zipCode);
  }
}
```

这部分代码太琐碎了，因此很容易搞乱，也很难调试。我们应该能把这种代码变得更优雅，为此，Guava提供了 `ComparisonChain` 。

ComparisonChain执行一种懒比较：它执行比较操作直至发现非零的结果，在那之后的比较输入将被忽略。

```
public int compareTo(Foo that) {
    return ComparisonChain.start()
            .compare(this.aString, that.aString)
            .compare(this.anInt, that.anInt)
            .compare(this.anEnum, that.anEnum, Ordering.natural(
).nullsLast())
            .result();
}
```

这种Fluent接口风格的可读性更高，发生错误编码的几率更小，并且能避免做不必要的工作。更多Guava排序器工具可以在下一节里找到。

1.4-排序: Guava强大的"流畅风格比较器"

原文链接 译者: 沈义扬

排序器[Ordering]是Guava流畅风格比较器[Comparator]的实现，它可以用来为构建复杂的比较器，以完成集合排序的功能。

从实现上说，Ordering实例就是一个特殊的Comparator实例。Ordering把很多基于Comparator的静态方法（如Collections.max）包装为自己的实例方法（非静态方法），并且提供了链式调用方法，来定制和增强现有的比较器。

创建排序器：常见的排序器可以由下面的静态方法创建

方法	描述
`natural()`	对可排序类型做自然排序，如数字按大小，日期按先后排序
`usingToString()`	按对象的字符串形式做字典排序[lexicographical ordering]
`from(Comparator)`	把给定的Comparator转化为排序器

实现自定义的排序器时，除了用上面的from方法，也可以跳过实现Comparator，而直接继承Ordering：

```
Ordering<String> byLengthOrdering = new Ordering<String>() {
  public int compare(String left, String right) {
    return Ints.compare(left.length(), right.length());
  }
};
```

链式调用方法：通过链式调用，可以由给定的排序器衍生出其它排序器

方法	描述
`reverse()`	获取语义相反的排序器
`nullsFirst()`	使用当前排序器，但额外把null值排到最前面。
`nullsLast()`	使用当前排序器，但额外把null值排到最后面。
`compound(Comparator)`	合成另一个比较器，以处理当前排序器中的相等情况。
`lexicographical()`	基于处理类型T的排序器，返回该类型的可迭代对象Iterable<T>的排序器。
`onResultOf(Function)`	对集合中元素调用Function，再按返回值用当前排序器排序。

例如，你需要下面这个类的排序器。

```
class Foo {
    @Nullable String sortedBy;
    int notSortedBy;
}
```

考虑到排序器应该能处理sortedBy为null的情况，我们可以使用下面的链式调用来合成排序器：

```
Ordering<Foo> ordering = Ordering.natural().nullsFirst().onResul
tOf(new Function<Foo, String>() {
  public String apply(Foo foo) {
    return foo.sortedBy;
  }
});
```

当阅读链式调用产生的排序器时，应该从后往前读。上面的例子中，排序器首先调用apply方法获取sortedBy值，并把sortedBy为null的元素都放到最前面，然后把剩下的元素按sortedBy进行自然排序。之所以要从后往前读，是因为每次链式调用都是用后面的方法包装了前面的排序器。

注：用*compound*方法包装排序器时，就不应遵循从后往前读的原则。为了避免理解上的混乱，请不要把*compound*写在一长串链式调用的中间，你可以另起一行，在链中最先或最后调用*compound*。

超过一定长度的链式调用，也可能会带来阅读和理解上的难度。我们建议按下面的代码这样，在一个链中最多使用三个方法。此外，你也可以把Function分离成中间对象，让链式调用更简洁紧凑。

```
Ordering<Foo> ordering = Ordering.natural().nullsFirst().onResul
tOf(sortKeyFunction)
```

运用排序器：Guava的排序器实现有若干操纵集合或元素值的方法

方法	描述
`greatestOf(Iterable iterable, int k)`	获取可迭代对象中最大的k个元素。
`isOrdered(Iterable)`	判断可迭代对象是否已按排序器排序：允许有排序值相等的元素。
`sortedCopy(Iterable)`	判断可迭代对象是否已严格按排序器排序：不允许排序值相等的元素。
`min(E, E)`	返回两个参数中最小的那个。如果相等，则返回第一个参数。
`min(E, E, E, E...)`	返回多个参数中最小的那个。如果有超过一个参数都最小，则返回第一个最小的参数。
`min(Iterable)`	返回迭代器中最小的元素。如果可迭代对象中没有元素，则抛出NoSuchElementException。

1.5-Throwables：简化异常和错误的传播与检查

原文链接 译者: 沈义扬

异常传播

有时候，你会想把捕获到的异常再次抛出。这种情况通常发生在Error或RuntimeException被捕获的时候，你没想捕获它们，但是声明捕获Throwable和Exception的时候，也包括了了Error或RuntimeException。Guava提供了若干方法，来判断异常类型并且重新传播异常。例如：

```
try {
    someMethodThatCouldThrowAnything();
} catch (IKnowWhatToDoWithThisException e) {
    handle(e);
} catch (Throwable t) {
    Throwables.propagateIfInstanceOf(t, IOException.class);
    Throwables.propagateIfInstanceOf(t, SQLException.class);
    throw Throwables.propagate(t);
}
```

所有这些方法都会自己决定是否要抛出异常，但也能直接抛出方法返回的结果——例如，throw Throwables.propagate(t);—— 这样可以向编译器声明这里一定会抛出异常。

Guava中的异常传播方法简要列举如下：

RuntimeException propagate(Throwable)	如果Throwable是Error或RuntimeException，直接抛出；否则把Throwable包装成RuntimeException抛出。返回类型是RuntimeException，所以你可以像上面说的那样写成 `throw Throwables.propagate(t)`，Java编译器会意识到这行代码保证抛出异常。
void propagateIfInstanceOf(Throwable, Class<X extends Exception>) throws X	Throwable类型为X才抛出
void propagateIfPossible(Throwable)	Throwable类型为Error或RuntimeException才抛出
void propagateIfPossible(Throwable, Class<X extends Throwable>) throws X	Throwable类型为X, Error或RuntimeException才抛出

Throwables.propagate的用法

模仿Java7的多重异常捕获和再抛出

通常来说，如果调用者想让异常传播到栈顶，他不需要写任何catch代码块。因为他不打算从异常中恢复，他可能就不应该记录异常，或者有其他的动作。他可能是想做一些清理工作，但通常来说，无论操作是否成功，清理工作都要进行，所以清理工作可能会放在finallly代码块中。但有时候，捕获异常然后再抛出也是有用的：也许调用者想要在异常传播之前统计失败的次数，或者有条件地传播异常。

当只对一种异常进行捕获和再抛出时，代码可能还是简单明了的。但当多种异常需要处理时，却可能变得一团糟：

```
@Override public void run() {
    try {
        delegate.run();
    } catch (RuntimeException e) {
        failures.increment();
        throw e;
    }catch (Error e) {
        failures.increment();
        throw e;
    }
}
```

Java7用多重捕获解决了这个问题：

```
    } catch (RuntimeException | Error e) {
        failures.increment();
        throw e;
    }
```

非Java7用户却受困于这个问题。他们想要写如下代码来统计所有异常，但是编译器不允许他们抛出Throwable（译者注：这种写法把原本是*Error*或*RuntimeException*类型的异常修改成了*Throwable*，因此调用者不得不修改方法签名）：

```
    } catch (Throwable t) {
        failures.increment();
        throw t;
    }
```

解决办法是用throw Throwables.propagate(t)替换throw t。在限定情况下（捕获Error和RuntimeException），Throwables.propagate和原始代码有相同行为。然而，用Throwables.propagate也很容易写出有其他隐藏行为的代码。尤其要注意的是，这个方案只适用于处理RuntimeException 或Error。如果catch块捕获了受检异常，你需要调用propagateIfInstanceOf来保留原始代码的行为，因为Throwables.propagate不能直接传播受检异常。

总之，Throwables.propagate的这种用法也就马马虎虎，在Java7中就没必要这样做了。在其他Java版本中，它可以减少少量的代码重复，但简单地提取方法进行重构也能做到这一点。此外，使用propagate会意外地包装受检异常。

非必要用法：把抛出的Throwable转为Exception

有少数API，尤其是Java反射API和（以此为基础的）Junit，把方法声明成抛出Throwable。和这样的API交互太痛苦了，因为即使是最通用的API通常也只是声明抛出Exception。当确定代码会抛出Throwable，而不是Exception或Error时，调用者可能会用Throwables.propagate转化Throwable。这里有个用Callable执行Junit测试的范例：

```
public Void call() throws Exception {
    try {
        FooTest.super.runTest();
    } catch (Throwable t) {
        Throwables.propagateIfPossible(t, Exception.class);
        Throwables.propagate(t);
    }

    return null;
}
```

在这儿没必要调用propagate()方法，因为propagateIfPossible传播了Throwable之外的所有异常类型，第二行的propagate就变得完全等价于throw new RuntimeException(t)。（题外话：这个例子也提醒我们，propagateIfPossible可能也会引起混乱，因为它不但会传播参数中给定的异常类型，还抛出Error和RuntimeException）

这种模式（或类似于throw new RuntimeException(t)的模式）在Google代码库中出现了超过30次。（搜索'propagateIfPossible'* Exception.class[]);'）绝大多数情况下都明确用了"throw new RuntimeException(t)"。我们也曾想过有个"throwWrappingWeirdThrowable"方法处理Throwable到Exception的转化。但考虑到我们用两行代码实现了这个模式，除非我们也丢弃propagateIfPossible方法，不然定义这个throwWrappingWeirdThrowable方法也并没有太大必要。

Throwables.propagate的有争议用法

争议一：把受检异常转化为非受检异常

原则上，非受检异常代表bug，而受检异常表示不可控的问题。但在实际运用中，即使JDK也有所误用——如Object.clone()、Integer. parseInt(String)、URI(String)——或者至少对某些方法来说，没有让每个人都信服的答案，如URI.create(String)的异常声明。

因此，调用者有时不得不把受检异常和非受检异常做相互转化：

```
try {
    return Integer.parseInt(userInput);
} catch (NumberFormatException e) {
    throw new InvalidInputException(e);
}
```

```
try {
    return publicInterfaceMethod.invoke();
} catch (IllegalAccessException e) {
    throw new AssertionError(e);
}
```

有时候，调用者会使用Throwables.propagate转化异常。这样做有没有什么缺点？最主要的恐怕是代码的含义不太明显。throw Throwables.propagate(ioException)做了什么？throw new RuntimeException(ioException)做了什么？这两者做了同样的事情，但后者的意思更简单直接。前者却引起了疑问："它做了什么？它并不只是把异常包装进RuntimeException吧？如果它真的只做了包装，为什么还非得要写个方法？"。应该承认，这些问题部分是因为"propagate"的语义太模糊了（用来抛出未声明的异常吗？）。也许"wrapIfChecked"更能清楚地表达含义。但即使方法叫做"wrapIfChecked"，用它来包装一个已知类型的受检异常也没什么优点。甚至会有

其他缺点：也许比起RuntimeException，还有更合适的类型——如IllegalArgumentException。 我们有时也会看到propagate被用于传播可能为受检的异常，结果是代码相比以前会稍微简短点，但也稍微有点不清晰：

```
} catch (RuntimeException e) {
    throw e;
}catch (Exception e) {
    throw new RuntimeException(e);
}
```

```
} catch (Exception e) {
 throw Throwables.propagate(e);
}
```

然而，我们似乎故意忽略了把检查型异常转化为非检查型异常的合理性。在某些场景中，这无疑是正确的做法，但更多时候它被用于避免处理受检异常。这让我们的话题变成了争论受检异常是不是坏主意了，我不想对此多做叙述。但可以这样说，Throwables.propagate不是为了鼓励开发者忽略IOException这样的异常。

争议二：异常穿隧

但是，如果你要实现不允许抛出异常的方法呢？有时候你需要把异常包装在非受检异常内。这种做法挺好，但我们再次强调，没必要用propagate方法做这种简单的包装。实际上，手动包装可能更好：如果你手动包装了所有异常（而不仅仅是受检异常），那你就可以在另一端解包所有异常，并处理极少数特殊场景。此外，你可能还想把异常包装成特定的类型，而不是像propagate这样统一包装成RuntimeException。

争议三：重新抛出其他线程产生的异常

```
try {
    return future.get();
} catch (ExecutionException e) {
    throw Throwables.propagate(e.getCause());
}
```

对这样的代码要考虑很多方面：

- ExecutionException的cause可能是受检异常，见上文"争议一：把检查型异常转化为非检查型异常"。但如果我们确定future对应的任务不会抛出受检异常呢？（可能future表示runnable任务的结果——译者注：如__*ExecutorService*中的*submit(Runnable task, T result)*方法）如上所述，你可以捕获异常并抛出AssertionError。尤其对于Future，请考虑 Futures.get方法。（TODO：对future.get()抛出的另一个异常InterruptedException作一些说明）

- ExecutionException的cause可能直接是Throwable类型，而不是Exception或Error。（实际上这不大可能，但你想直接重新抛出cause的话，编译器会强迫你考虑这种可能性）见上文"用法二：把抛出Throwable改为抛出Exception"。
- ExecutionException的cause可能是非受检异常。如果是这样的话，cause会直接被Throwables.propagate抛出。不幸的是，cause的堆栈信息反映的是异常最初产生的线程，而不是传播异常的线程。通常来说，最好在异常链中同时包含这两个线程的堆栈信息，就像ExecutionException所做的那样。（这个问题并不单单和propagate方法相关；所有在其他线程中重新抛出异常的代码都需要考虑这点）

异常原因链

Guava提供了如下三个有用的方法，让研究异常的原因链变得稍微简便了，这三个方法的签名是不言自明的：

Throwable getRootCause(Throwable)
List<Throwable> getCausalChain(Throwable)
String getStackTraceAsString(Throwable)

2-集合

2.1-不可变集合

原文链接 译者：沈义扬

范例

```
public static final ImmutableSet<String> COLOR_NAMES = Immutable
Set.of(
        "red",
        "orange",
        "yellow",
        "green",
        "blue",
        "purple");

class Foo {
    Set<Bar> bars;
    Foo(Set<Bar> bars) {
        this.bars = ImmutableSet.copyOf(bars); // defensive copy
!
    }
}
```

为什么要使用不可变集合

不可变对象有很多优点，包括：

- 当对象被不可信的库调用时，不可变形式是安全的；
- 不可变对象被多个线程调用时，不存在竞态条件问题
- 不可变集合不需要考虑变化，因此可以节省时间和空间。所有不可变的集合都比它们的可变形式有更好的内存利用率（分析和测试细节）；
- 不可变对象因为有固定不变，可以作为常量来安全使用。

创建对象的不可变拷贝是一项很好的防御性编程技巧。Guava为所有JDK标准集合类型和Guava新集合类型都提供了简单易用的不可变版本。 JDK也提供了Collections.unmodifiableXXX方法把集合包装为不可变形式，但我们认为不够好：

- 笨重而且累赘：不能舒适地用在所有想做防御性拷贝的场景；
- 不安全：要保证没人通过原集合的引用进行修改，返回的集合才是事实上不可变的；
- 低效：包装过的集合仍然保有可变集合的开销，比如并发修改的检查、散列表的额外空间，等等。

如果你没有修改某个集合的需求，或者希望某个集合保持不变时，把它防御性地拷贝到不可变集合是个很好的实践。

重要提示：所有*Guava*不可变集合的实现都不接受*null*值。我们对*Google*内部的代码库做过详细研究，发现只有*5%*的情况需要在集合中允许*null*元素，剩下的*95%*场景都是遇到*null*值就快速失败。如果你需要在不可变集合中使用*null*，请使用*JDK*中的*Collections.unmodifiableXXX*方法。更多细节建议请参考__*"使用和避免null"*。

怎么使用不可变集合

不可变集合可以用如下多种方式创建：

- copyOf方法，如ImmutableSet.copyOf(set);
- of方法，如ImmutableSet.of("a", "b", "c")或 ImmutableMap.of("a", 1, "b", 2);
- Builder工具，如

```
public static final ImmutableSet<Color> GOOGLE_COLORS =
        ImmutableSet.<Color>builder()
            .addAll(WEBSAFE_COLORS)
            .add(new Color(0, 191, 255))
            .build();
```

此外，对有序不可变集合来说，排序是在构造集合的时候完成的，如：

```
ImmutableSortedSet.of("a", "b", "c", "a", "d", "b");
```

会在构造时就把元素排序为a, b, c, d。

比想象中更智能的copyOf

请注意，ImmutableXXX.copyOf方法会尝试在安全的时候避免做拷贝——实际的实现细节不详，但通常来说是很智能的，比如：

```
ImmutableSet<String> foobar = ImmutableSet.of("foo", "bar", "baz
");
thingamajig(foobar);

void thingamajig(Collection<String> collection) {
    ImmutableList<String> defensiveCopy = ImmutableList.copyOf(c
ollection);
    ...
}
```

在这段代码中，ImmutableList.copyOf(foobar)会智能地直接返回foobar.asList(),它是一个ImmutableSet的常量时间复杂度的List视图。 作为一种探索，ImmutableXXX.copyOf(ImmutableCollection)会试图对如下情况避免线性时间拷贝：

- 在常量时间内使用底层数据结构是可能的——例如，ImmutableSet.copyOf(ImmutableList)就不能在常量时间内完成。
- 不会造成内存泄露——例如，你有个很大的不可变集合ImmutableList<String> hugeList， ImmutableList.copyOf(hugeList.subList(0, 10))就会显式地拷贝，以免不必要地持有hugeList的引用。
- 不改变语义——所以ImmutableSet.copyOf(myImmutableSortedSet)会显式地拷贝，因为和基于比较器的ImmutableSortedSet相比，ImmutableSet对hashCode()和equals有不同语义。

在可能的情况下避免线性拷贝，可以最大限度地减少防御性编程风格所带来的性能开销。

asList视图

所有不可变集合都有一个asList()方法提供ImmutableList视图，来帮助你用列表形式方便地读取集合元素。例如，你可以使用sortedSet.asList().get(k)从ImmutableSortedSet中读取第k个最小元素。

asList()返回的ImmutableList通常是——并不总是——开销稳定的视图实现，而不是简单地把元素拷贝进List。也就是说，asList返回的列表视图通常比一般的列表平均性能更好，比如，在底层集合支持的情况下，它总是使用高效的contains方法。

细节：关联可变集合和不可变集合

可变集合接口	属于**JDK还是Guava**	不可变版本
Collection	JDK	`ImmutableCollection`
List	JDK	`ImmutableList`
Set	JDK	`ImmutableSet`
SortedSet/NavigableSet	JDK	`ImmutableSortedSet`
Map	JDK	`ImmutableMap`
SortedMap	JDK	`ImmutableSortedMap`
Multiset	Guava	`ImmutableMultiset`
SortedMultiset	Guava	`ImmutableSortedMultiset`
Multimap	Guava	`ImmutableMultimap`
ListMultimap	Guava	`ImmutableListMultimap`
SetMultimap	Guava	`ImmutableSetMultimap`
BiMap	Guava	`ImmutableBiMap`
ClassToInstanceMap	Guava	`ImmutableClassToInstanceMap`
Table	Guava	`ImmutableTable`

2.2-新集合类型

原文链接 译文链接 译者：沈义扬，校对：丁一

Guava引入了很多JDK没有的、但我们发现明显有用的新集合类型。这些新类型是为了和JDK集合框架共存，而没有往JDK集合抽象中硬塞其他概念。作为一般规则，Guava集合非常精准地遵循了JDK接口契约。

Multiset

统计一个词在文档中出现了多少次，传统的做法是这样的：

```
Map<String, Integer> counts = new HashMap<String, Integer>();
for (String word : words) {
    Integer count = counts.get(word);
    if (count == null) {
        counts.put(word, 1);
    } else {
        counts.put(word, count + 1);
    }
}
```

这种写法很笨拙，也容易出错，并且不支持同时收集多种统计信息，如总词数。我们可以做的更好。

Guava提供了一个新集合类型 Multiset，它可以多次添加相等的元素。维基百科从数学角度这样定义Multiset："集合[set]概念的延伸，它的元素可以重复出现…与集合[set]相同而与元组[tuple]相反的是，Multiset元素的顺序是无关紧要的：Multiset {a, a, b}和{a, b, a}是相等的"。——译者注：这里所说的集合_*[set]*是数学上的概念，*Multiset*继承自*JDK*中的*Collection*接口，而不是*Set*接口，所以包含重复元素并没有违反原有的接口契约。

可以用两种方式看待Multiset：

- 没有元素顺序限制的ArrayList<E>
- Map<E, Integer>，键为元素，值为计数

Guava的Multiset API也结合考虑了这两种方式： 当把Multiset看成普通的Collection时，它表现得就像无序的ArrayList：

- add(E)添加单个元素
- iterator()返回一个迭代器，包含Multiset的所有元素（包括重复的元素）
- size()返回所有元素的总个数（包括重复的元素）

当把Multiset看作Map<E, Integer>时，它也提供了符合性能期望的查询操作：

- count(Object)返回给定元素的计数。HashMultiset.count的复杂度为O(1)，

TreeMultiset.count的复杂度为O(log n)。
- entrySet()返回Set<Multiset.Entry<E>>，和Map的entrySet类似。
- elementSet()返回所有不重复元素的Set<E>，和Map的keySet()类似。
- 所有Multiset实现的内存消耗随着不重复元素的个数线性增长。

值得注意的是，除了极少数情况，Multiset和JDK中原有的Collection接口契约完全一致——具体来说，TreeMultiset在判断元素是否相等时，与TreeSet一样用compare，而不是Object.equals。另外特别注意，Multiset.addAll(Collection)可以添加Collection中的所有元素并进行计数，这比用for循环往Map添加元素和计数方便多了。

方法	描述
count(E)	给定元素在Multiset中的计数
elementSet()	Multiset中不重复元素的集合，类型为Set<E>
entrySet()	和Map的entrySet类似，返回Set<Multiset.Entry<E>>，其中包含的Entry支持getElement()和getCount()方法
add(E, int)	增加给定元素在Multiset中的计数
remove(E, int)	减少给定元素在Multiset中的计数
setCount(E, int)	设置给定元素在Multiset中的计数，不可以为负数
size()	返回集合元素的总个数（包括重复的元素）

Multiset不是Map

请注意，Multiset<E>不是Map<E, Integer>，虽然Map可能是某些Multiset实现的一部分。准确来说Multiset是一种Collection类型，并履行了Collection接口相关的契约。关于Multiset和Map的显著区别还包括：

- Multiset中的元素计数只能是正数。任何元素的计数都不能为负，也不能是0。elementSet()和entrySet()视图中也不会有这样的元素。
- multiset.size()返回集合的大小，等同于所有元素计数的总和。对于不重复元素的个数，应使用elementSet().size()方法。（因此，add(E)把multiset.size()增加1）
- multiset.iterator()会迭代重复元素，因此迭代长度等于multiset.size()。
- Multiset支持直接增加、减少或设置元素的计数。setCount(elem, 0)等同于移除所有elem。
- 对multiset 中没有的元素，multiset.count(elem)始终返回0。

Multiset的各种实现

Guava提供了多种Multiset的实现，大致对应JDK中Map的各种实现：

Map	对应的**Multiset**	是否支持**null**元素
HashMap	HashMultiset	是
TreeMap	TreeMultiset	是（如果comparator支持的话）
LinkedHashMap	LinkedHashMultiset	是
ConcurrentHashMap	ConcurrentHashMultiset	否
ImmutableMap	ImmutableMultiset	否

SortedMultiset

SortedMultiset是Multiset 接口的变种，它支持高效地获取指定范围的子集。比方说，你可以用 latencies.subMultiset(0,BoundType.CLOSED, 100, BoundType.OPEN).size()来统计你的站点中延迟在100毫秒以内的访问，然后把这个值和latencies.size()相比，以获取这个延迟水平在总体访问中的比例。

TreeMultiset实现SortedMultiset接口。在撰写本文档时，ImmutableSortedMultiset还在测试和GWT的兼容性。

Multimap

每个有经验的Java程序员都在某处实现过Map<K, List<V>>或Map<K, Set<V>>，并且要忍受这个结构的笨拙。例如，Map<K, Set<V>>通常用来表示非标定有向图。Guava的 Multimap可以很容易地把一个键映射到多个值。换句话说，Multimap是把键映射到任意多个值的一般方式。

可以用两种方式思考Multimap的概念：”键-单个值映射”的集合：

```
a -> 1 a -> 2 a ->4 b -> 3 c -> 5
```

或者”键-值集合映射”的映射：

```
a -> [1, 2, 4] b -> 3 c -> 5
```

一般来说，Multimap接口应该用第一种方式看待，但asMap()视图返回Map<K, Collection<V>>，让你可以按另一种方式看待Multimap。重要的是，不会有任何键映射到空集合：一个键要么至少到一个值，要么根本就不在Multimap中。

很少会直接使用Multimap接口，更多时候你会用ListMultimap或SetMultimap接口，它们分别把键映射到List或Set。

修改Multimap

Multimap.get(key)以集合形式返回键所对应的值视图，即使没有任何对应的值，也会返回空集合。ListMultimap.get(key)返回List，SetMultimap.get(key)返回Set。

对值视图集合进行的修改最终都会反映到底层的Multimap。例如：

```
Set<Person> aliceChildren = childrenMultimap.get(alice);
aliceChildren.clear();
aliceChildren.add(bob);
aliceChildren.add(carol);
```

其他（更直接地）修改Multimap的方法有：

方法签名	描述	等价于
put(K, V)	添加键到单个值的映射	multimap.get(key).add(value)
putAll(K, Iterable<V>)	依次添加键到多个值的映射	Iterables.addAll(multimap.get(key), values)
remove(K, V)	移除键到值的映射；如果有这样的键值并成功移除，返回true。	multimap.get(key).remove(value)
removeAll(K)	清除键对应的所有值，返回的集合包含所有之前映射到K的值，但修改这个集合就不会影响Multimap了。	multimap.get(key).clear()
replaceValues(K, Iterable<V>)	清除键对应的所有值，并重新把key关联到Iterable中的每个元素。返回的集合包含所有之前映射到K的值。	multimap.get(key).clear(); Iterables.addAll(multimap.get(key), values)

Multimap的视图

Multimap还支持若干强大的视图：

- `asMap` 为Multimap<K, V>提供Map<K,Collection<V>>形式的视图。返回的Map支持remove操作，并且会反映到底层的Multimap，但它不支持put或putAll操作。更重要的是，如果你想为Multimap中没有的键返回null，而不是一个新的、可写的空集合，你就可以使用asMap().get(key)。（你可以并且应当把asMap.get(key)返回的结果转化为适当的集合类型——如SetMultimap.asMap.get(key)的结果转为Set，ListMultimap.asMap.get(key)的

结果转为List——Java类型系统不允许ListMultimap直接为asMap.get(key)返回List——译者注：也可以用__*Multimaps*中的*asMap*静态方法帮你完成类型转换）

- `entries` 用Collection<Map.Entry<K, V>>返回Multimap中所有"键-单个值映射"——包括重复键。（对SetMultimap，返回的是Set）
- `keySet` 用Set表示Multimap中所有不同的键。
- `keys` 用Multiset表示Multimap中的所有键，每个键重复出现的次数等于它映射的值的个数。可以从这个Multiset中移除元素，但不能做添加操作；移除操作会反映到底层的Multimap。
- `values()` 用一个"扁平"的Collection<V>包含Multimap中的所有值。这有一点类似于Iterables.concat(multimap.asMap().values())，但它直接返回了单个Collection，而不像multimap.asMap().values()那样是按键区分开的Collection。

Multimap不是Map

Multimap<K, V>不是Map<K,Collection<V>>，虽然某些Multimap实现中可能使用了map。它们之间的显著区别包括：

- Multimap.get(key)总是返回非null、但是可能空的集合。这并不意味着Multimap为相应的键花费内存创建了集合，而只是提供一个集合视图方便你为键增加映射值——译者注：如果有这样的键，返回的集合只是包装了__*Multimap*中已有的集合；如果没有这样的键，返回的空集合也只是持有*Multimap*引用的栈对象，让你可以用来操作底层的*Multimap*。因此，返回的集合不会占据太多内存，数据实际上还是存放在*Multimap*中。
- 如果你更喜欢像Map那样，为Multimap中没有的键返回null，请使用asMap()视图获取一个Map<K, Collection<V>>。（或者用静态方法Multimaps.asMap()为ListMultimap返回一个Map<K, List<V>>。对于SetMultimap和SortedSetMultimap，也有类似的静态方法存在）
- 当且仅当有值映射到键时，Multimap.containsKey(key)才会返回true。尤其需要注意的是，如果键k之前映射过一个或多个值，但它们都被移除后，Multimap.containsKey(key)会返回false。
- Multimap.entries()返回Multimap中所有"键-单个值映射"——包括重复键。如果你想要得到所有"键-值集合映射"，请使用asMap().entrySet()。
- Multimap.size()返回所有"键-单个值映射"的个数，而非不同键的个数。要得到不同键的个数，请改用Multimap.keySet().size()。

Multimap的各种实现

Multimap提供了多种形式的实现。在大多数要使用Map<K, Collection<V>>的地方，你都可以使用它们：

实现	键行为类似	值行为类似
ArrayListMultimap	HashMap	ArrayList
HashMultimap	HashMap	HashSet
LinkedListMultimap*	LinkedHashMap*	LinkedList*
LinkedHashMultimap**	LinkedHashMap	LinkedHashMap
TreeMultimap	TreeMap	TreeSet
`ImmutableListMultimap`	ImmutableMap	ImmutableList
ImmutableSetMultimap	ImmutableMap	ImmutableSet

除了两个不可变形式的实现，其他所有实现都支持null键和null值

*LinkedListMultimap.entries()保留了所有键和值的迭代顺序。详情见doc链接。

**LinkedHashMultimap保留了映射项的插入顺序，包括键插入的顺序，以及键映射的所有值的插入顺序。

请注意，并非所有的Multimap都和上面列出的一样，使用Map<K, Collection<V>>来实现（特别是，一些Multimap实现用了自定义的hashTable，以最小化开销）

如果你想要更大的定制化，请用Multimaps.newMultimap(Map, Supplier<Collection>)或list和 set版本，使用自定义的Collection、List或Set实现Multimap。

BiMap

传统上，实现键值对的双向映射需要维护两个单独的map，并保持它们间的同步。但这种方式很容易出错，而且对于值已经在map中的情况，会变得非常混乱。例如：

```
Map<String, Integer> nameToId = Maps.newHashMap();
Map<Integer, String> idToName = Maps.newHashMap();

nameToId.put("Bob", 42);
idToName.put(42, "Bob");
//如果"Bob"和42已经在map中了，会发生什么?
//如果我们忘了同步两个map，会有诡异的bug发生...
```

BiMap<K, V>是特殊的Map：

- 可以用 inverse()反转BiMap<K, V>的键值映射
- 保证值是唯一的，因此 values()返回Set而不是普通的Collection

在BiMap中，如果你想把键映射到已经存在的值，会抛出IllegalArgumentException异常。如果对特定值，你想要强制替换它的键，请使用 BiMap.forcePut(key, value)。

```
BiMap<String, Integer> userId = HashBiMap.create();
...

String userForId = userId.inverse().get(id);
```

BiMap的各种实现

键**–**值实现	值**–**键实现	对应的**BiMap**实现
HashMap	HashMap	HashBiMap
ImmutableMap	ImmutableMap	ImmutableBiMap
EnumMap	EnumMap	EnumBiMap
EnumMap	HashMap	EnumHashBiMap

注：Maps类中还有一些诸如synchronizedBiMap的BiMap工具方法.

Table

```
Table<Vertex, Vertex, Double> weightedGraph = HashBasedTable.cre
ate();
weightedGraph.put(v1, v2, 4);
weightedGraph.put(v1, v3, 20);
weightedGraph.put(v2, v3, 5);

weightedGraph.row(v1); // returns a Map mapping v2 to 4, v3 to 2
0
weightedGraph.column(v3); // returns a Map mapping v1 to 20, v2
to 5
```

通常来说，当你想使用多个键做索引的时候，你可能会用类似Map<FirstName, Map<LastName, Person>>的实现，这种方式很丑陋，使用上也不友好。Guava为此提供了新集合类型Table，它有两个支持所有类型的键："行"和"列"。Table提供多种视图，以便你从各种角度使用它：

- rowMap()：用Map<R, Map<C, V>>表现Table<R, C, V>。同样的，rowKeySet()返回"行"的集合Set<R>。
- row(r) ：用Map<C, V>返回给定"行"的所有列，对这个map进行的写操作也将写入Table中。
- 类似的列访问方法：columnMap()、columnKeySet()、column(c)。（基于列的访问会比基于的行访问稍微低效点）

- cellSet()：用元素类型为Table.Cell<R, C, V>的Set表现Table<R, C, V>。Cell类似于Map.Entry，但它是用行和列两个键区分的。

Table有如下几种实现：

- HashBasedTable：本质上用HashMap<R, HashMap<C, V>>实现；
- TreeBasedTable：本质上用TreeMap<R, TreeMap<C,V>>实现；
- ImmutableTable：本质上用ImmutableMap<R, ImmutableMap<C, V>>实现；注：ImmutableTable对稀疏或密集的数据集都有优化。
- ArrayTable：要求在构造时就指定行和列的大小，本质上由一个二维数组实现，以提升访问速度和密集Table的内存利用率。ArrayTable与其他Table的工作原理有点不同，请参见Javadoc了解详情。

ClassToInstanceMap

ClassToInstanceMap是一种特殊的Map：它的键是类型，而值是符合键所指类型的对象。

为了扩展Map接口，ClassToInstanceMap额外声明了两个方法：T getInstance(Class<T>) 和T putInstance(Class<T>, T)，从而避免强制类型转换，同时保证了类型安全。

ClassToInstanceMap有唯一的泛型参数，通常称为B，代表Map支持的所有类型的上界。例如：

```
ClassToInstanceMap<Number> numberDefaults=MutableClassToInstance
Map.create();
numberDefaults.putInstance(Integer.class, Integer.valueOf(0));
```

从技术上讲，ClassToInstanceMap<B>实现了Map<Class<? extends B>, B>——或者换句话说，是一个映射B的子类型到对应实例的Map。这让ClassToInstanceMap包含的泛型声明有点令人困惑，但请记住B始终是Map所支持类型的上界——通常B就是Object。

对于ClassToInstanceMap，Guava提供了两种有用的实现：MutableClassToInstanceMap和 ImmutableClassToInstanceMap。

RangeSet

RangeSet描述了一组不相连的、非空的区间。当把一个区间添加到可变的RangeSet时，所有相连的区间会被合并，空区间会被忽略。例如：

```
RangeSet<Integer> rangeSet = TreeRangeSet.create();
rangeSet.add(Range.closed(1, 10)); // {[1,10]}
rangeSet.add(Range.closedOpen(11, 15));//不相连区间:{[1,10], [11,1
5)}
rangeSet.add(Range.closedOpen(15, 20)); //相连区间; {[1,10], [11,
20)}
rangeSet.add(Range.openClosed(0, 0)); //空区间; {[1,10], [11,20)}
rangeSet.remove(Range.open(5, 10)); //分割[1, 10]; {[1,5], [10,10
], [11,20)}
```

请注意，要合并Range.closed(1, 10)和Range.closedOpen(11, 15)这样的区间，你需要首先用Range.canonical(DiscreteDomain)对区间进行预处理，例如DiscreteDomain.integers()。

注：RangeSet不支持GWT，也不支持JDK5和更早版本；因为，RangeSet需要充分利用JDK6中NavigableMap的特性。

RangeSet的视图

RangeSet的实现支持非常广泛的视图：

- complement()：返回RangeSet的补集视图。complement也是RangeSet类型，包含了不相连的、非空的区间。
- subRangeSet(Range<C>)：返回RangeSet与给定Range的交集视图。这扩展了传统排序集合中的headSet、subSet和tailSet操作。
- asRanges()：用Set<Range<C>>表现RangeSet，这样可以遍历其中的Range。
- asSet(DiscreteDomain<C>)（仅ImmutableRangeSet支持）：用ImmutableSortedSet<C>表现RangeSet，以区间中所有元素的形式而不是区间本身的形式查看。（这个操作不支持DiscreteDomain 和RangeSet都没有上边界，或都没有下边界的情况）

RangeSet的查询方法

为了方便操作，RangeSet直接提供了若干查询方法，其中最突出的有:

- contains(C)：RangeSet最基本的操作，判断RangeSet中是否有任何区间包含给定元素。
- rangeContaining(C)：返回包含给定元素的区间；若没有这样的区间，则返回null。
- encloses(Range<C>)：简单明了，判断RangeSet中是否有任何区间包括给定区间。
- span()：返回包括RangeSet中所有区间的最小区间。

RangeMap

RangeMap描述了”不相交的、非空的区间”到特定值的映射。和RangeSet不同，RangeMap不会合并相邻的映射，即便相邻的区间映射到相同的值。例如：

```
RangeMap<Integer, String> rangeMap = TreeRangeMap.create();
rangeMap.put(Range.closed(1, 10), "foo"); //{[1,10] => "foo"}
rangeMap.put(Range.open(3, 6), "bar"); //{[1,3] => "foo", (3,6)
=> "bar", [6,10] => "foo"}
rangeMap.put(Range.open(10, 20), "foo"); //{[1,3] => "foo", (3,6
) => "bar", [6,10] => "foo", (10,20) => "foo"}
rangeMap.remove(Range.closed(5, 11)); //{[1,3] => "foo", (3,5) =
> "bar", (11,20) => "foo"}
```

RangeMap的视图

RangeMap提供两个视图：

- asMapOfRanges()：用Map<Range<K>, V>表现RangeMap。这可以用来遍历RangeMap。
- subRangeMap(Range<K>)：用RangeMap类型返回RangeMap与给定Range的交集视图。这扩展了传统的headMap、subMap和tailMap操作。

2.3-强大的集合工具类：java.util.Collections中未包含的集合工具

原文链接 译文链接 译者：沈义扬，校对：丁一

尚未完成: *Queues, Tables*工具类

任何对JDK集合框架有经验的程序员都熟悉和喜欢 `java.util.Collections` 包含的工具方法。Guava沿着这些路线提供了更多的工具方法：适用于所有集合的静态方法。这是Guava最流行和成熟的部分之一。

我们用相对直观的方式把工具类与特定集合接口的对应关系归纳如下：

集合接口	属于**JDK还是Guava**	对应的**Guava**工具类
Collection	JDK	`Collections2` ：不要和java.util.Collections混淆
List	JDK	`Lists`
Set	JDK	`Sets`
SortedSet	JDK	`Sets`
Map	JDK	`Maps`
SortedMap	JDK	`Maps`
Queue	JDK	`Queues`
Multiset	Guava	`Multisets`
Multimap	Guava	`Multimaps`
BiMap	Guava	`Maps`
Table	Guava	`Tables`

在找类似转化、过滤的方法？请看第四章，函数式风格。

静态工厂方法

在JDK 7之前，构造新的范型集合时要讨厌地重复声明范型：

```
List<TypeThatsTooLongForItsOwnGood> list = new ArrayList<TypeThatsTooLongForItsOwnGood>();
```

我想我们都认为这很讨厌。因此Guava提供了能够推断范型的静态工厂方法：

```
List<TypeThatsTooLongForItsOwnGood> list = Lists.newArrayList();
Map<KeyType, LongishValueType> map = Maps.newLinkedHashMap();
```

可以肯定的是，JDK7版本的钻石操作符(<>)没有这样的麻烦：

```
List<TypeThatsTooLongForItsOwnGood> list = new ArrayList<>();
```

但Guava的静态工厂方法远不止这么简单。用工厂方法模式，我们可以方便地在初始化时就指定起始元素。

```
Set<Type> copySet = Sets.newHashSet(elements);
List<String> theseElements = Lists.newArrayList("alpha", "beta",
 "gamma");
```

此外，通过为工厂方法命名（Effective Java第一条），我们可以提高集合初始化大小的可读性：

```
List<Type> exactly100 = Lists.newArrayListWithCapacity(100);
List<Type> approx100 = Lists.newArrayListWithExpectedSize(100);
Set<Type> approx100Set = Sets.newHashSetWithExpectedSize(100);
```

确切的静态工厂方法和相应的工具类一起罗列在下面的章节。

注意：Guava引入的新集合类型没有暴露原始构造器，也没有在工具类中提供初始化方法。而是直接在集合类中提供了静态工厂方法，例如：

```
Multiset<String> multiset = HashMultiset.create();
```

Iterables

在可能的情况下，Guava提供的工具方法更偏向于接受Iterable而不是Collection类型。在Google，对于不存放在主存的集合——比如从数据库或其他数据中心收集的结果集，因为实际上还没有攫取全部数据，这类结果集都不能支持类似size()的操作 ——通常都不会用Collection类型来表示。

因此，很多你期望的支持所有集合的操作都在 `Iterables` 类中。大多数Iterables方法有一个在Iterators类中的对应版本，用来处理Iterator。

截至Guava 1.2版本，Iterables使用 `[FluentIterable](http://docs.guava-libraries.googlecode.com/git-h` 进行了补充，它包装了一个Iterable实例，并对许多操作提供了"fluent"（链式调

用）语法。

下面列出了一些最常用的工具方法，但更多Iterables的函数式方法将在第四章讨论。

常规方法

concat(Iterable<Iterable>)	串联多个iterables的懒视图*	conc
frequency(Iterable, Object)	返回对象在iterable中出现的次数	与Col
partition(Iterable, int)	把iterable按指定大小分割，得到的子集都不能进行修改操作	List
getFirst(Iterable, T default)	返回iterable的第一个元素，若iterable为空则返回默认值	与Iter
getLast(Iterable)	返回iterable的最后一个元素，若iterable为空则抛出NoSuchElementException	getL
elementsEqual(Iterable, Iterable)	如果两个iterable中的所有元素相等且顺序一致，返回true	与Lis
unmodifiableIterable(Iterable)	返回iterable的不可变视图	与Col
limit(Iterable, int)	限制iterable的元素个数限制给定值	Flue
getOnlyElement(Iterable)	获取iterable中唯一的元素，如果iterable为空或有多个元素，则快速失败	get(

*译者注：懒视图意味着如果还没访问到某个*iterable*中的元素，则不会对它进行串联操作。

```
Iterable<Integer> concatenated = Iterables.concat(
        Ints.asList(1, 2, 3),
        Ints.asList(4, 5, 6)); // concatenated包括元素 1, 2, 3, 4
, 5, 6
String lastAdded = Iterables.getLast(myLinkedHashSet);
String theElement = Iterables.getOnlyElement(thisSetIsDefinitely
ASingleton);
//如果set不是单元素集，就会出错了！
```

与Collection方法相似的工具方法

通常来说，Collection的实现天然支持操作其他Collection，但却不能操作Iterable。

下面的方法中，如果传入的Iterable是一个Collection实例，则实际操作将会委托给相应的Collection接口方法。例如，往Iterables.size方法传入是一个Collection实例，它不会真的遍历iterator获取大小，而是直接调用Collection.size。

方法	类似的**Coll
`addAll(Collection addTo, Iterable toAdd)`	Collection.ad
`contains(Iterable, Object)`	Collection.co
`removeAll(Iterable removeFrom, Collection toRemove)`	Collection.rer
`retainAll(Iterable removeFrom, Collection toRetain)`	Collection.ret
`size(Iterable)`	Collection.siz
`toArray(Iterable, Class)`	Collection.toA
`isEmpty(Iterable)`	Collection.isE
`get(Iterable, int)`	List.get(int)
`toString(Iterable)`	Collection.toS

FluentIterable

除了上面和第四章提到的方法，FluentIterable还有一些便利方法用来把自己拷贝到不可变集合

ImmutableList	
ImmutableSet	`toImmutableSet()`
ImmutableSortedSet	`toImmutableSortedSet(Comparator)`

Lists

除了静态工厂方法和函数式编程方法， `Lists` 为List类型的对象提供了若干工具方法。

方法	描述
`partition(List, int)`	把List按指定大小分割
`reverse(List)`	返回给定List的反转视图。注: 如果List是不可变的，考虑改用 `ImmutableList.reverse()` 。

```
List countUp = Ints.asList(1, 2, 3, 4, 5);
List countDown = Lists.reverse(theList); // {5, 4, 3, 2, 1}
List<List> parts = Lists.partition(countUp, 2);//{{1,2}, {3,4},
{5}}
```

静态工厂方法

Lists提供如下静态工厂方法：

具体实现类型	工厂方法
ArrayList	basic, with elements, from `Iterable`, with exact capacity, with expected size, from `Iterator`
LinkedList	basic, from `Iterable`

Sets

`Sets` 工具类包含了若干好用的方法。

集合理论方法

我们提供了很多标准的集合运算（Set-Theoretic）方法，这些方法接受Set参数并返回 `SetView` ，可用于：

- 直接当作Set使用，因为SetView也实现了Set接口；
- 用 `copyInto(Set)` 拷贝进另一个可变集合；
- 用 `immutableCopy()` 对自己做不可变拷贝。

方法
`union(Set, Set)`
`intersection(Set, Set)`
`difference(Set, Set)`
`symmetricDifference(Set, Set)`

使用范例：

```
Set<String> wordsWithPrimeLength = ImmutableSet.of("one", "two",
 "three", "six", "seven", "eight");
Set<String> primes = ImmutableSet.of("two", "three", "five", "se
ven");
SetView<String> intersection = Sets.intersection(primes,wordsWit
hPrimeLength);
// intersection包含"two", "three", "seven"
return intersection.immutableCopy();//可以使用交集，但不可变拷贝的读
取效率更高
```

其他Set工具方法

方法	描述	另请参见
cartesianProduct(List<Set>)	返回所有集合的笛卡儿积	cartesianProduct(Set...
powerSet(Set)	返回给定集合的所有子集	

```
Set<String> animals = ImmutableSet.of("gerbil", "hamster");
Set<String> fruits = ImmutableSet.of("apple", "orange", "banana"
);

Set<List<String>> product = Sets.cartesianProduct(animals, fruit
s);
// {{"gerbil", "apple"}, {"gerbil", "orange"}, {"gerbil", "banan
a"},
//  {"hamster", "apple"}, {"hamster", "orange"}, {"hamster", "ba
nana"}}

Set<Set<String>> animalSets = Sets.powerSet(animals);
// {{}, {"gerbil"}, {"hamster"}, {"gerbil", "hamster"}}
```

静态工厂方法

Sets提供如下静态工厂方法：

具体实现类型	工厂方法
HashSet	basic, with elements, from `Iterable`, with expected size, from `Iterator`
LinkedHashSet	basic, from `Iterable`, with expected size
TreeSet	basic, with `Comparator`, from `Iterable`

Maps

`Maps` 类有若干值得单独说明的、很酷的方法。

uniqueIndex

`Maps.uniqueIndex(Iterable,Function)` 通常针对的场景是：有一组对象，它们在某个属性上分别有独一无二的值，而我们希望能够按照这个属性值查找对象——译者注：这个方法返回一个__*Map*，键为*Function*返回的属性值，值为*Iterable*中相应的元素，因此我们可以反复用这个*Map*进行查找操作。

比方说，我们有一堆字符串，这些字符串的长度都是独一无二的，而我们希望能够按照特定长度查找字符串：

```
ImmutableMap<Integer, String> stringsByIndex = Maps.uniqueIndex(
strings,
    new Function<String, Integer> () {
        public Integer apply(String string) {
            return string.length();
        }
    });
```

如果索引值不是独一无二的，请参见下面的Multimaps.index方法。

difference

`Maps.difference(Map, Map)` 用来比较两个Map以获取所有不同点。该方法返回MapDifference对象，把不同点的维恩图分解为：

`entriesInCommon()`	两个**Map**中都有的映射项，包括匹配的键与值
`entriesDiffering()`	键相同但是值不同值映射项。返回的Map的值类型为 `MapDifference.ValueDifference` ，以表示左右两个不同的值
`entriesOnlyOnLeft()`	键只存在于左边Map的映射项
`entriesOnlyOnRight()`	键只存在于右边Map的映射项

```
Map<String, Integer> left = ImmutableMap.of("a", 1, "b", 2, "c",
 3);
Map<String, Integer> left = ImmutableMap.of("a", 1, "b", 2, "c",
 3);
MapDifference<String, Integer> diff = Maps.difference(left, righ
t);

diff.entriesInCommon(); // {"b" => 2}
diff.entriesInCommon(); // {"b" => 2}
diff.entriesOnlyOnLeft(); // {"a" => 1}
diff.entriesOnlyOnRight(); // {"d" => 5}
```

处理BiMap的工具方法

Guava中处理BiMap的工具方法在Maps类中，因为BiMap也是一种Map实现。

BiMap**工具方法**	相应的**Map**工具方法
`synchronizedBiMap(BiMap)`	Collections.synchronizedMap(Map)
`unmodifiableBiMap(BiMap)`	Collections.unmodifiableMap(Map)

静态工厂方法

Maps提供如下静态工厂方法：

具体实现类型	工厂方法
HashMap	basic, from `Map`, with expected size
LinkedHashMap	basic, from `Map`
TreeMap	basic, from `Comparator`, from `SortedMap`
EnumMap	from `Class`, from `Map`
ConcurrentMap：支持所有操作	basic
IdentityHashMap	basic

Multisets

标准的Collection操作会忽略Multiset重复元素的个数，而只关心元素是否存在于Multiset中，如containsAll方法。为此，`Multisets`提供了若干方法，以顾及Multiset元素的重复性：

方法	说明
`containsOccurrences(Multiset sup, Multiset sub)`	对任意 sub.co <=sup 回true
`removeOccurrences(Multiset removeFrom, Multiset toRemove)`	对toR 元素， remov 相同个
`retainOccurrences(Multiset removeFrom, Multiset toRetain)`	修改re 以保证 remov <=toR
`intersection(Multiset, Multiset)`	返回两 集;

```
Multiset<String> multiset1 = HashMultiset.create();
multiset1.add("a", 2);

Multiset<String> multiset2 = HashMultiset.create();
multiset2.add("a", 5);

multiset1.containsAll(multiset2); //返回true；因为包含了所有不重复元素，
//虽然multiset1实际上包含2个"a"，而multiset2包含5个"a"
Multisets.containsOccurrences(multiset1, multiset2); // returns false

multiset2.removeOccurrences(multiset1); // multiset2 现在包含3个"a"
multiset2.removeAll(multiset1);//multiset2移除所有"a"，虽然multiset1只有2个"a"
multiset2.isEmpty(); // returns true
```

Multisets中的其他工具方法还包括：

`copyHighestCountFirst(Multiset)`	返回**Multiset**的不可变拷贝，并将元素按重复出现的次数做降序排列
`unmodifiableMultiset(Multiset)`	返回Multiset的只读视图
`unmodifiableSortedMultiset(SortedMultiset)`	返回SortedMultiset的只读视图

```
Multiset<String> multiset = HashMultiset.create();
multiset.add("a", 3);
multiset.add("b", 5);
multiset.add("c", 1);

ImmutableMultiset highestCountFirst = Multisets.copyHighestCountFirst(multiset);
//highestCountFirst，包括它的entrySet和elementSet，按{"b", "a", "c"}排列元素
```

Multimaps

`Multimaps`提供了若干值得单独说明的通用工具方法

index

作为Maps.uniqueIndex的兄弟方法，`Multimaps.index(Iterable, Function)`通常针对的场景是：有一组对象，它们有共同的特定属性，我们希望按照这个属性的值查询对象，但属性值不一定是独一无二的。

比方说，我们想把字符串按长度分组。

```
ImmutableSet digits = ImmutableSet.of("zero", "one", "two", "thr
ee", "four", "five", "six", "seven", "eight", "nine");
Function<String, Integer> lengthFunction = new Function<String,
Integer>() {
    public Integer apply(String string) {
        return string.length();
    }
};

ImmutableListMultimap<Integer, String> digitsByLength= Multimaps
.index(digits, lengthFunction);
/*
*   digitsByLength maps:
*   3 => {"one", "two", "six"}
*   4 => {"zero", "four", "five", "nine"}
*   5 => {"three", "seven", "eight"}
*/
```

invertFrom

鉴于Multimap可以把多个键映射到同一个值（译者注：实际上这是任何__*map*都有的特性），也可以把一个键映射到多个值，反转Multimap也会很有用。Guava 提供了`invertFrom(Multimap toInvert, Multimap dest)`)做这个操作，并且你可以自由选择反转后的Multimap实现。

注：如果你使用的是ImmutableMultimap，考虑改用`ImmutableMultimap.inverse()`做反转。

```
ArrayListMultimap<String, Integer> multimap = ArrayListMultimap.
create();
multimap.putAll("b", Ints.asList(2, 4, 6));
multimap.putAll("a", Ints.asList(4, 2, 1));
multimap.putAll("c", Ints.asList(2, 5, 3));

TreeMultimap<Integer, String> inverse = Multimaps.invertFrom(mul
timap, TreeMultimap<String, Integer>.create());
//注意我们选择的实现，因为选了TreeMultimap，得到的反转结果是有序的
/*
* inverse maps:
*  1 => {"a"}
*  2 => {"a", "b", "c"}
*  3 => {"c"}
*  4 => {"a", "b"}
*  5 => {"c"}
*  6 => {"b"}
*/
```

forMap

想在Map对象上使用Multimap的方法吗？`forMap(Map)`把Map包装成SetMultimap。这个方法特别有用，例如，与Multimaps.invertFrom结合使用，可以把多对一的Map反转为一对多的Multimap。

```
Map<String, Integer> map = ImmutableMap.of("a", 1, "b", 1, "c",
2);
SetMultimap<String, Integer> multimap = Multimaps.forMap(map);
// multimap：["a" => {1}, "b" => {1}, "c" => {2}]
Multimap<Integer, String> inverse = Multimaps.invertFrom(multima
p, HashMultimap<Integer, String>.create());
// inverse：[1 => {"a","b"}, 2 => {"c"}]
```

包装器

Multimaps提供了传统的包装方法，以及让你选择Map和Collection类型以自定义Multimap实现的工具方法。

只读包装	`Multimap`	`ListMultimap`	`SetMultimap`	`SortedSetMultima`
同步包装	`Multimap`	`ListMultimap`	`SetMultimap`	`SortedSetMultima`
自定义实现	`Multimap`	`ListMultimap`	`SetMultimap`	`SortedSetMultima`

自定义Multimap的方法允许你指定Multimap中的特定实现。但要注意的是：

- Multimap假设对Map和Supplier产生的集合对象有完全所有权。这些自定义对象应避免手动更新，并且在提供给Multimap时应该是空的，此外还不应该使用软引用、弱引用或虚引用。
- 无法保证修改了Multimap以后，底层Map的内容是什么样的。
- 即使Map和Supplier产生的集合都是线程安全的，它们组成的Multimap也不能保证并发操作的线程安全性。并发读操作是工作正常的，但需要保证并发读写的话，请考虑用同步包装器解决。
- 只有当Map、Supplier、Supplier产生的集合对象、以及Multimap存放的键值类型都是可序列化的，Multimap才是可序列化的。
- Multimap.get(key)返回的集合对象和Supplier返回的集合对象并不是同一类型。但如果Supplier返回的是随机访问集合，那么Multimap.get(key)返回的集合也是可随机访问的。

请注意，用来自定义Multimap的方法需要一个Supplier参数，以创建崭新的集合。下面有个实现ListMultimap的例子——用TreeMap做映射，而每个键对应的多个值用LinkedList存储。

```
ListMultimap<String, Integer> myMultimap = Multimaps.newListMult
imap(
    Maps.<String, Collection>newTreeMap(),
    new Supplier<LinkedList>() {
        public LinkedList get() {
            return Lists.newLinkedList();
        }
    });
```

Tables

`Tables` 类提供了若干称手的工具方法。

自定义Table

堪比Multimaps.newXXXMultimap(Map, Supplier)工具方法，Tables.newCustomTable(Map, Supplier<Map>)允许你指定Table用什么样的map实现行和列。

```
// 使用LinkedHashMaps替代HashMaps
Table<String, Character, Integer> table = Tables.newCustomTable(
Maps.<String, Map<Character, Integer>>newLinkedHashMap(),
new Supplier<Map<Character, Integer>> () {
public Map<Character, Integer> get() {
return Maps.newLinkedHashMap();
}
});
```

transpose

transpose(Table<R, C, V>)方法允许你把Table<C, R, V>转置成Table<R, C, V>。例如，如果你在用Table构建加权有向图，这个方法就可以把有向图反转。

包装器

还有很多你熟悉和喜欢的Table包装类。然而，在大多数情况下还请使用 ImmutableTable

Unmodifiable
Table
RowSortedTable

2.4-集合扩展工具类

原文链接 译文链接 译者：沈义扬，校对：丁一

简介

有时候你需要实现自己的集合扩展。也许你想要在元素被添加到列表时增加特定的行为，或者你想实现一个Iterable，其底层实际上是遍历数据库查询的结果集。Guava为你，也为我们自己提供了若干工具方法，以便让类似的工作变得更简单。（毕竟，我们自己也要用这些工具扩展集合框架。）

Forwarding装饰器

针对所有类型的集合接口，Guava都提供了Forwarding抽象类以简化装饰者模式的使用。

Forwarding抽象类定义了一个抽象方法：delegate()，你可以覆盖这个方法来返回被装饰对象。所有其他方法都会直接委托给delegate()。例如说：ForwardingList.get(int)实际上执行了delegate().get(int)。

通过创建ForwardingXXX的子类并实现delegate()方法，可以选择性地覆盖子类的方法来增加装饰功能，而不需要自己委托每个方法——译者注：因为所有方法都默认委托给__*delegate()*返回的对象，你可以只覆盖需要装饰的方法。

此外，很多集合方法都对应一个"标准方法[standardxxx]"实现，可以用来恢复被装饰对象的默认行为，以提供相同的优点。比如在扩展AbstractList或JDK中的其他骨架类时，可以使用类似standardAddAll这样的方法。

让我们看看这个例子。假定你想装饰一个List，让其记录所有添加进来的元素。当然，无论元素是用什么方法——add(int, E), add(E), 或addAll(Collection)——添加进来的，我们都希望进行记录，因此我们需要覆盖所有这些方法。

```
class AddLoggingList<E> extends ForwardingList<E> {
    final List<E> delegate; // backing list
    @Override protected List<E> delegate() {
        return delegate;
    }
    @Override public void add(int index, E elem) {
        log(index, elem);
        super.add(index, elem);
    }
    @Override public boolean add(E elem) {
        return standardAdd(elem); // 用add(int, E)实现
    }
    @Override public boolean addAll(Collection<? extends E> c) {
        return standardAddAll(c); // 用add实现
    }
}
```

记住，默认情况下，所有方法都直接转发到被代理对象，因此覆盖ForwardingMap.put并不会改变ForwardingMap.putAll的行为。小心覆盖所有需要改变行为的方法，并且确保装饰后的集合满足接口契约。

通常来说，类似于AbstractList的抽象集合骨架类，其大多数方法在Forwarding装饰器中都有对应的"标准方法"实现。

对提供特定视图的接口，Forwarding装饰器也为这些视图提供了相应的"标准方法"实现。例如，ForwardingMap提供StandardKeySet、StandardValues和StandardEntrySet类，它们在可以的情况下都会把自己的方法委托给被装饰的Map，把不能委托的声明为抽象方法。

PeekingIterator

有时候，普通的Iterator接口还不够。

Iterators提供一个 `Iterators.peekingIterator(Iterator)` 方法，来把Iterator包装为 `PeekingIterator` ，这是Iterator的子类，它能让你事先窥视[`peek()`]到下一次调用next()返回的元素。

注意：Iterators.peekingIterator返回的PeekingIterator不支持在peek()操作之后调用remove()方法。

举个例子：复制一个List，并去除连续的重复元素。

```
List<E> result = Lists.newArrayList();
PeekingIterator<E> iter = Iterators.peekingIterator(source.itera
tor());
while (iter.hasNext()) {
    E current = iter.next();
    while (iter.hasNext() && iter.peek().equals(current)) {
        //跳过重复的元素
        iter.next();
    }
    result.add(current);
}
```

传统的实现方式需要记录上一个元素，并在特定情况下后退，但这很难处理且容易出错。相较而言，PeekingIterator在理解和使用上就比较直接了。

AbstractIterator

实现你自己的Iterator？ `AbstractIterator` 让生活更轻松。

用一个例子来解释AbstractIterator最简单。比方说，我们要包装一个iterator以跳过空值。

```
public static Iterator<String> skipNulls(final Iterator<String>
in) {
    return new AbstractIterator<String>() {
        protected String computeNext() {
            while (in.hasNext()) {
                String s = in.next();
                if (s != null) {
                    return s;
                }
            }
            return endOfData();
        }
    };
}
```

你实现了 `computeNext()` 方法，来计算下一个值。如果循环结束了也没有找到下一个值，请返回endOfData()表明已经到达迭代的末尾。

注意：AbstractIterator继承了UnmodifiableIterator，所以禁止实现remove()方法。如果你需要支持remove()的迭代器，就不应该继承AbstractIterator。

AbstractSequentialIterator

有一些迭代器用其他方式表示会更简单。 `AbstractSequentialIterator` 就提供了表示迭代的另一种方式。

```
Iterator<Integer> powersOfTwo = new AbstractSequentialIterator<I
nteger>(1) { // 注意初始值1!
    protected Integer computeNext(Integer previous) {
        return (previous == 1 << 30) ? null : previous * 2;
    }
};
```

我们在这儿实现了`computeNext(T)`方法，它能接受前一个值作为参数。

注意，你必须额外传入一个初始值，或者传入null让迭代立即结束。因为computeNext(T)假定null值意味着迭代的末尾——AbstractSequentialIterator不能用来实现可能返回null的迭代器。

3-缓存

原文地址 译文地址 译者：许巧辉 校对：沈义扬

范例

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .maximumSize(1000)
        .expireAfterWrite(10, TimeUnit.MINUTES)
        .removalListener(MY_LISTENER)
        .build(
            new CacheLoader<Key, Graph>() {
                public Graph load(Key key) throws AnyException {
                    return createExpensiveGraph(key);
                }
        });
```

适用性

缓存在很多场景下都是相当有用的。例如，计算或检索一个值的代价很高，并且对同样的输入需要不止一次获取值的时候，就应当考虑使用缓存。 缓存在很多场景下都是相当有用的。例如，计算或检索一个值的代价很高，并且对同样的输入需要不止一次获取值的时候，就应当考虑使用缓存。

Guava Cache与ConcurrentMap很相似，但也不完全一样。最基本的区别是ConcurrentMap会一直保存所有添加的元素，直到显式地移除。相对地，Guava Cache为了限制内存占用，通常都设定为自动回收元素。在某些场景下，尽管LoadingCache 不回收元素，它也是很有用的，因为它会自动加载缓存。

通常来说，`Guava Cache` 适用于：

- 你愿意消耗一些内存空间来提升速度。
- 你预料到某些键会被查询一次以上。
- 缓存中存放的数据总量不会超出内存容量。（Guava Cache是单个应用运行时的本地缓存。它不把数据存放到文件或外部服务器。如果这不符合你的需求，请尝试Memcached这类工具）

如果你的场景符合上述的每一条，Guava Cache就适合你。

如同范例代码展示的一样，Cache实例通过CacheBuilder生成器模式获取，但是自定义你的缓存才是最有趣的部分。

注：如果你不需要Cache中的特性，使用ConcurrentHashMap有更好的内存效率——但Cache的大多数特性都很难基于旧有的ConcurrentMap复制，甚至根本不可能做到。

加载

在使用缓存前，首先问自己一个问题：有没有合理的默认方法来加载或计算与键关联的值？如果有的话，你应当使用CacheLoader。如果没有，或者你想要覆盖默认的加载运算，同时保留"获取缓存-如果没有-则计算"[get-if-absent-compute]的原子语义，你应该在调用get时传入一个Callable实例。缓存元素也可以通过Cache.put方法直接插入，但自动加载是首选的，因为它可以更容易地推断所有缓存内容的一致性。

CacheLoader

LoadingCache是附带CacheLoader构建而成的缓存实现。创建自己的CacheLoader通常只需要简单地实现V load(K key) throws Exception方法。例如，你可以用下面的代码构建LoadingCache：

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .maximumSize(1000)
        .build(
            new CacheLoader<Key, Graph>() {
                public Graph load(Key key) throws AnyException {
                    return createExpensiveGraph(key);
                }
            });

...
try {
    return graphs.get(key);
} catch (ExecutionException e) {
    throw new OtherException(e.getCause());
}
```

从LoadingCache查询的正规方式是使用get(K)方法。这个方法要么返回已经缓存的值，要么使用CacheLoader向缓存原子地加载新值。由于CacheLoader可能抛出异常，LoadingCache.get(K)也声明为抛出ExecutionException异常。如果你定义的CacheLoader没有声明任何检查型异常，则可以通过getUnchecked(K)查找缓存；但必须注意，一旦CacheLoader声明了检查型异常，就不可以调用getUnchecked(K)。

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .expireAfterAccess(10, TimeUnit.MINUTES)
        .build(
            new CacheLoader<Key, Graph>() {
                public Graph load(Key key) { // no checked excep
tion
                    return createExpensiveGraph(key);
                }
            });

...
return graphs.getUnchecked(key);
```

getAll(Iterable<? extends K>)方法用来执行批量查询。默认情况下，对每个不在缓存中的键，getAll方法会单独调用CacheLoader.load来加载缓存项。如果批量的加载比多个单独加载更高效，你可以重载CacheLoader.loadAll来利用这一点。getAll(Iterable)的性能也会相应提升。

注：*CacheLoader.loadAll*的实现可以为没有明确请求的键加载缓存值。例如，为某组中的任意键计算值时，能够获取该组中的所有键值，*loadAll*方法就可以实现为在同一时间获取该组的其他键值。校注：*getAll(Iterable<? extends K>)*方法会调用*loadAll*，但会筛选结果，只会返回请求的键值对。

Callable

所有类型的Guava Cache，不管有没有自动加载功能，都支持get(K, Callable<V>)方法。这个方法返回缓存中相应的值，或者用给定的Callable运算并把结果加入到缓存中。在整个加载方法完成前，缓存项相关的可观察状态都不会更改。这个方法简便地实现了模式"如果有缓存则返回；否则运算、缓存、然后返回"。

```
Cache<Key, Graph> cache = CacheBuilder.newBuilder()
        .maximumSize(1000)
        .build(); // look Ma, no CacheLoader
...
try {
    // If the key wasn't in the "easy to compute" group, we need
 to
    // do things the hard way.
    cache.get(key, new Callable<Key, Graph>() {
        @Override
        public Value call() throws AnyException {
            return doThingsTheHardWay(key);
        }
    });
} catch (ExecutionException e) {
    throw new OtherException(e.getCause());
}
```

显式插入

使用cache.put(key, value)方法可以直接向缓存中插入值，这会直接覆盖掉给定键之前映射的值。使用Cache.asMap()视图提供的任何方法也能修改缓存。但请注意，asMap视图的任何方法都不能保证缓存项被原子地加载到缓存中。进一步说，asMap视图的原子运算在Guava Cache的原子加载范畴之外，所以相比于Cache.asMap().putIfAbsent(K, V)，Cache.get(K, Callable<V>) 应该总是优先使用。

缓存回收

一个残酷的现实是，我们几乎一定没有足够的内存缓存所有数据。你你必须决定：什么时候某个缓存项就不值得保留了？Guava Cache提供了三种基本的缓存回收方式：基于容量回收、定时回收和基于引用回收。

基于容量的回收（**size-based eviction**）

如果要规定缓存项的数目不超过固定值，只需使用 `CacheBuilder.maximumSize(long)` 。缓存将尝试回收最近没有使用或总体上很少使用的缓存项。——警告：在缓存项的数目达到限定值之前，缓存就可能进行回收操作——通常来说，这种情况发生在缓存项的数目逼近限定值时。

另外，不同的缓存项有不同的“权重”（weights）——例如，如果你的缓存值，占据完全不同的内存空间，你可以使用 `CacheBuilder.weigher(Weigher)` 指定一个权重函数，并且用 `CacheBuilder.maximumWeight(long)` 指定最大总重。在权重限定场景中，除了要注意回收也是在重量逼近限定值时就进行了，还要知道重量是在缓存创建时计算的，因此要考虑重量计算的复杂度。

```
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .maximumWeight(100000)
        .weigher(new Weigher<Key, Graph>() {
            public int weigh(Key k, Graph g) {
                return g.vertices().size();
            }
        })
        .build(
            new CacheLoader<Key, Graph>() {
                public Graph load(Key key) { // no checked excep
tion
                    return createExpensiveGraph(key);
                }
            });
```

定时回收（**Timed Eviction**）

`CacheBuilder``提供两种定时回收的方法：`

- `expireAfterAccess(long, TimeUnit)`：缓存项在给定时间内没有被读/写访问，则回收。请注意这种缓存的回收顺序和基于大小回收一样。
- `expireAfterWrite(long, TimeUnit)`：缓存项在给定时间内没有被写访问（创建或覆盖），则回收。如果认为缓存数据总是在固定时候后变得陈旧不可用，这种回收方式是可取的。

如下文所讨论，定时回收周期性地在写操作中执行，偶尔在读操作中执行。

测试定时回收

对定时回收进行测试时，不一定非得花费两秒钟去测试两秒的过期。你可以使用 `Ticker` 接口和 `CacheBuilder.ticker(Ticker)` 方法在缓存中自定义一个时间源，而不是非得用系统时钟。

基于引用的回收（**Reference-based Eviction**）

通过使用弱引用的键、或弱引用的值、或软引用的值，Guava Cache可以把缓存设置为允许垃圾回收：

- `CacheBuilder.weakKeys()`：使用弱引用存储键。当键没有其它（强或软）引用时，缓存项可以被垃圾回收。因为垃圾回收仅依赖恒等式（==），使用弱引用键的缓存用==而不是equals比较键。
- `CacheBuilder.weakValues()`：使用弱引用存储值。当值没有其它（强或软）引用时，缓存项可以被垃圾回收。因为垃圾回收仅依赖恒等式（==），使用弱引用值的缓存用==而不是equals比较值。
- `CacheBuilder.softValues()`：使用软引用存储值。软引用只有在响应内存需要时，才按照全局最近最少使用的顺序回收。考虑到使用软引用的性能影响，我们通常建议使用更有性能预测性的缓存大小限定（见上文，基于容量回收）。使用软引用值的缓存同样用==而不是equals比较值。

显式清除

任何时候，你都可以显式地清除缓存项，而不是等到它被回收：

- 个别清除：`Cache.invalidate(key)`
- 批量清除：`Cache.invalidateAll(keys)`
- 清除所有缓存项：`Cache.invalidateAll()`

移除监听器

通过 `CacheBuilder.removalListener(RemovalListener)`，你可以声明一个监听器，以便缓存项被移除时做一些额外操作。缓存项被移除时，`RemovalListener` 会获取移除通知 `RemovalNotification`，其中包含移除原因 `RemovalCause`、键和值。

请注意，RemovalListener抛出的任何异常都会在记录到日志后被丢弃[swallowed]。

```
CacheLoader<Key, DatabaseConnection> loader = new CacheLoader<Ke
y, DatabaseConnection> () {
    public DatabaseConnection load(Key key) throws Exception {
        return openConnection(key);
    }
};

RemovalListener<Key, DatabaseConnection> removalListener = new R
emovalListener<Key, DatabaseConnection>() {
    public void onRemoval(RemovalNotification<Key, DatabaseConne
ction> removal) {
        DatabaseConnection conn = removal.getValue();
        conn.close(); // tear down properly
    }
};

return CacheBuilder.newBuilder()
    .expireAfterWrite(2, TimeUnit.MINUTES)
    .removalListener(removalListener)
    .build(loader);
```

警告：默认情况下，监听器方法是在移除缓存时同步调用的。因为缓存的维护和请求响应通常是同时进行的，代价高昂的监听器方法在同步模式下会拖慢正常的缓存请求。在这种情况下，你可以使用 `RemovalListeners.asynchronous(RemovalListener, Executor)` 把监听器装饰为异步操作。

清理什么时候发生？

使用CacheBuilder构建的缓存不会"自动"执行清理和回收工作，也不会在某个缓存项过期后马上清理，也没有诸如此类的清理机制。相反，它会在写操作时顺带做少量的维护工作，或者偶尔在读操作时做——如果写操作实在太少的话。

这样做的原因在于：如果要自动地持续清理缓存，就必须有一个线程，这个线程会和用户操作竞争共享锁。此外，某些环境下线程创建可能受限制，这样CacheBuilder就不可用了。

相反，我们把选择权交到你手里。如果你的缓存是高吞吐的，那就无需担心缓存的维护和清理等工作。如果你的 缓存只会偶尔有写操作，而你又不想清理工作阻碍了读操作，那么可以创建自己的维护线程，以固定的时间间隔调用 `Cache.cleanUp()` 。 `ScheduledExecutorService` 可以帮助你很好地实现这样的定时调度。

刷新

刷新和回收不太一样。正如LoadingCache.refresh(K)所声明，刷新表示为键加载新值，这个过程可以是异步的。在刷新操作进行时，缓存仍然可以向其他线程返回旧值，而不像回收操作，读缓存的线程必须等待新值加载完成。

如果刷新过程抛出异常，缓存将保留旧值，而异常会在记录到日志后被丢弃[swallowed]。

重载CacheLoader.reload(K, V)可以扩展刷新时的行为，这个方法允许开发者在计算新值时使用旧的值。

```
//有些键不需要刷新，并且我们希望刷新是异步完成的
LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
        .maximumSize(1000)
        .refreshAfterWrite(1, TimeUnit.MINUTES)
        .build(
            new CacheLoader<Key, Graph>() {
                public Graph load(Key key) { // no checked excep
tion
                    return getGraphFromDatabase(key);
                }

                public ListenableFuture<Key, Graph> reload(final
 Key key, Graph prevGraph) {
                    if (neverNeedsRefresh(key)) {
                        return Futures.immediateFuture(prevGraph
);
                    }else{
                        // asynchronous!
                        ListenableFutureTask<Key, Graph> task=Li
stenableFutureTask.create(new Callable<Key, Graph>() {
                            public Graph call() {
                                return getGraphFromDatabase(key)
;
                            }
                        });
                        executor.execute(task);
                        return task;
                    }
                }
            });
```

CacheBuilder.refreshAfterWrite(long, TimeUnit)可以为缓存增加自动定时刷新功能。和expireAfterWrite相反，refreshAfterWrite通过定时刷新可以让缓存项保持可用，但请注意：缓存项只有在被检索时才会真正刷新（如果CacheLoader.refresh实现为异步，那么检索不会被刷新拖慢）。因此，如果你在缓存上同时声明expireAfterWrite和refreshAfterWrite，缓存并不会因为刷新盲目地定时重置，如果缓存项没有被检索，那刷新就不会真的发生，缓存项在过期时间后也变得可以回收。

其他特性

统计

`CacheBuilder.recordStats()` 用来开启Guava Cache的统计功能。统计打开后，`Cache.stats()` 方法会返回 `CacheStats` 对象以提供如下统计信息：

- `hitRate()`：缓存命中率；
- `averageLoadPenalty()`：加载新值的平均时间，单位为纳秒；
- `evictionCount()`：缓存项被回收的总数，不包括显式清除。

此外，还有其他很多统计信息。这些统计信息对于调整缓存设置是至关重要的，在性能要求高的应用中我们建议密切关注这些数据。

asMap视图

asMap视图提供了缓存的ConcurrentMap形式，但asMap视图与缓存的交互需要注意：

- cache.asMap()包含当前所有加载到缓存的项。因此相应地，cache.asMap().keySet()包含当前所有已加载键;
- asMap().get(key)实质上等同于cache.getIfPresent(key)，而且不会引起缓存项的加载。这和Map的语义约定一致。
- 所有读写操作都会重置相关缓存项的访问时间，包括Cache.asMap().get(Object)方法和Cache.asMap().put(K, V)方法，但不包括Cache.asMap().containsKey(Object)方法，也不包括在Cache.asMap()的集合视图上的操作。比如，遍历Cache.asMap().entrySet()不会重置缓存项的读取时间。

中断

缓存加载方法（如Cache.get）不会抛出InterruptedException。我们也可以让这些方法支持InterruptedException，但这种支持注定是不完备的，并且会增加所有使用者的成本，而只有少数使用者实际获益。详情请继续阅读。

Cache.get请求到未缓存的值时会遇到两种情况：当前线程加载值；或等待另一个正在加载值的线程。这两种情况下的中断是不一样的。等待另一个正在加载值的线程属于较简单的情况：使用可中断的等待就实现了中断支持；但当前线程加载值的情况就比较复杂了：因为加载值的CacheLoader是由用户提供的，如果它是可中断的，那我们也可以实现支持中断，否则我们也无能为力。

如果用户提供的CacheLoader是可中断的，为什么不让Cache.get也支持中断？从某种意义上说，其实是支持的：如果CacheLoader抛出InterruptedException，Cache.get将立刻返回（就和其他异常情况一样）；此外，在加载缓存值的线程中，Cache.get捕捉到InterruptedException后将恢复中断，而其他线程中InterruptedException则被包装成了ExecutionException。

原则上，我们可以拆除包装，把ExecutionException变为InterruptedException，但这会让所有的LoadingCache使用者都要处理中断异常，即使他们提供的CacheLoader不是可中断的。如果你考虑到所有非加载线程的等待仍可以被中断，

这种做法也许是值得的。但许多缓存只在单线程中使用，它们的用户仍然必须捕捉不可能抛出的InterruptedException异常。即使是那些跨线程共享缓存的用户，也只是有时候能中断他们的get调用，取决于那个线程先发出请求。

对于这个决定，我们的指导原则是让缓存始终表现得好像是在当前线程加载值。这个原则让使用缓存或每次都计算值可以简单地相互切换。如果老代码（加载值的代码）是不可中断的，那么新代码（使用缓存加载值的代码）多半也应该是不可中断的。

如上所述，Guava Cache在某种意义上支持中断。另一个意义上说，Guava Cache不支持中断，这使得LoadingCache成了一个有漏洞的抽象：当加载过程被中断了，就当作其他异常一样处理，这在大多数情况下是可以的；但如果多个线程在等待加载同一个缓存项，即使加载线程被中断了，它也不应该让其他线程都失败（捕获到包装在ExecutionException里的InterruptedException），正确的行为是让剩余的某个线程重试加载。为此，我们记录了一个bug。然而，与其冒着风险修复这个bug，我们可能会花更多的精力去实现另一个建议AsyncLoadingCache，这个实现会返回一个有正确中断行为的Future对象。

4-函数式编程

原文链接 译文链接 译者：沈义扬，校对：丁一

注意事项

截至JDK7，Java中也只能通过笨拙冗长的匿名类来达到近似函数式编程的效果。预计JDK8中会有所改变，但Guava现在就想给JDK5以上用户提供这类支持。

过度使用Guava函数式编程会导致冗长、混乱、可读性差而且低效的代码。这是迄今为止最容易（也是最经常）被滥用的部分，如果你想通过函数式风格达成一行代码，致使这行代码长到荒唐，Guava团队会泪流满面。

比较如下代码：

```
Function<String, Integer> lengthFunction = new Function<String,
Integer>() {
    public Integer apply(String string) {
        return string.length();
    }
};
Predicate<String> allCaps = new Predicate<String>() {
    public boolean apply(String string) {
        return CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string);
    }
};
Multiset<Integer> lengths = HashMultiset.create(
     Iterables.transform(Iterables.filter(strings, allCaps), len
gthFunction));
```

或FluentIterable的版本

```
Multiset<Integer> lengths = HashMultiset.create(
    FluentIterable.from(strings)
        .filter(new Predicate<String>() {
            public boolean apply(String string) {
                return CharMatcher.JAVA_UPPER_CASE.matchesAllOf(
string);
            }
        })
        .transform(new Function<String, Integer>() {
            public Integer apply(String string) {
                return string.length();
            }
        }));
```

还有

```
Multiset<Integer> lengths = HashMultiset.create();
for (String string : strings) {
    if (CharMatcher.JAVA_UPPER_CASE.matchesAllOf(string)) {
        lengths.add(string.length());
    }
}
```

即使用了静态导入，甚至把Function和Predicate的声明放到别的文件，第一种代码实现仍然不简洁，可读性差并且效率较低。

截至JDK7，命令式代码仍应是默认和第一选择。不应该随便使用函数式风格，除非你绝对确定以下两点之一：

- 使用函数式风格以后，整个工程的代码行会净减少。在上面的例子中，函数式版本用了11行， 命令式代码用了6行，把函数的定义放到另一个文件或常量中，并不能帮助减少总代码行。
- 为了提高效率，转换集合的结果需要懒视图，而不是明确计算过的集合。此外，确保你已经阅读和重读了Effective Java的第55条，并且除了阅读本章后面的说明，你还真正做了性能测试并且有测试数据来证明函数式版本更快。

请务必确保，当使用Guava函数式的时候，用传统的命令式做同样的事情不会更具可读性。尝试把代码写下来，看看它是不是真的那么糟糕？会不会比你想尝试的极其笨拙的函数式 更具可读性。

Functions[函数]和Predicates[断言]

本节只讨论直接与Function和Predicate打交道的Guava功能。一些其他工具类也和"函数式风格"相关，例如 `Iterables.concat(Iterable<Iterable>)` ，和其他用常量时间返回视图的方法。尝试看看2.3节的集合工具类。

Guava提供两个基本的函数式接口：

- Function<A, B>，它声明了单个方法B apply(A input)。Function对象通常被预期为引用透明的——没有副作用——并且引用透明性中的"相等"语义与equals一致，如a.equals(b)意味着function.apply(a).equals(function.apply(b))。
- Predicate<T>，它声明了单个方法boolean apply(T input)。Predicate对象通常也被预期为无副作用函数，并且"相等"语义与equals一致。

特殊的断言

字符类型有自己特定版本的Predicate—— `CharMatcher` ，它通常更高效，并且在某些需求方面更有用。CharMatcher实现了Predicate<Character>，可以当作Predicate一样使用，要把Predicate转成CharMatcher，可以使用 `CharMatcher.forPredicate` 。更多细节请参考第6章-字符串处理。

此外，对可比较类型和基于比较逻辑的Predicate，Range类可以满足大多数需求——它表示一个不可变区间。Range类实现了Predicate，用以判断值是否在区间内。例如，Range.atMost(2)就是个完全合法的Predicate<Integer>。更多使用Range的细节请参照第8章。

操作Functions和Predicates

Functions提供简便的Function构造和操作方法，包括：

forMap(Map<A, B>)	compose(Function<B, C>, Function
identity()	toStringFunction()

细节请参考Javadoc。

相应地，Predicates提供了更多构造和处理Predicate的方法，下面是一些例子：

instanceOf(Class)	assignableFrom(Class)	contains(Pattern)
in(Collection)	isNull()	alwaysFalse()
alwaysTrue()	equalTo(Object)	compose(Predicate,
and(Predicate...)	or(Predicate...)	not(Predicate)

细节请参考Javadoc。

使用函数式编程

Guava提供了很多工具方法，以便用Function或Predicate操作集合。这些方法通常可以在集合工具类找到，如Iterables，Lists，Sets，Maps，Multimaps等。

断言

断言的最基本应用就是过滤集合。所有Guava过滤方法都返回"视图"——译者注：即并非用一个新的集合表示过滤，而只是基于原集合的视图。

集合类型	过滤方法
Iterable	`Iterables.filter(Iterable, Predicate)` `FluentIterable.`
Iterator	`Iterators.filter(Iterator, Predicate)`
Collection	`Collections2.filter(Collection, Predicate)`
Set	`Sets.filter(Set, Predicate)`
SortedSet	`Sets.filter(SortedSet, Predicate)`
Map	`Maps.filterKeys(Map, Predicate)` `Maps.filterValues(Map`
SortedMap	`Maps.filterKeys(SortedMap, Predicate)` `Maps.filterValu`
Multimap	`Multimaps.filterKeys(Multimap, Predicate)` `Multimaps.f`

*List的过滤视图被省略了，因为不能有效地支持类似get(int)的操作。请改用Lists.newArrayList(Collections2.filter(list, predicate))做拷贝过滤。

除了简单过滤，Guava另外提供了若干用Predicate处理Iterable的工具——通常在 `Iterables` 工具类中，或者是 `FluentIterable` 的"fluent"（链式调用）方法。

Iterables**方法签名**	说明
`boolean all(Iterable, Predicate)`	是否所有元素满足断 懒实现：如果发现有 不满足，不会继续迭
`boolean any(Iterable, Predicate)`	是否有任意元素满足 满足断言？懒实现： 迭代到发现满足的元
`T find(Iterable, Predicate)`	循环并返回一个满足 满足断言的元素，如 有则抛出 NoSuchElementExc
`Optional<T> tryFind(Iterable, Predicate)`	返回一个满足元素满 言的元素，若没有则 回 o ptional.absen
`indexOf(Iterable, Predicate)`	返回第一个满足元素 断言的元素索引值， 有返回-1
`removeIf(Iterable, Predicate)`	移除所有满足元素满 言的元素，实际调用 Iterator.remove()方

函数

到目前为止，函数最常见的用途为转换集合。同样，所有的Guava转换方法也返回原集合的视图。

集合类型	转换**方法**
Iterable	Iterables.transform(Iterable, Function) FluentItera
Iterator	Iterators.transform(Iterator, Function)
Collection	Collections2.transform(Collection, Function)
List	Lists.transform(List, Function)
Map*	Maps.transformValues(Map, Function) Maps.transformE
SortedMap*	Maps.transformValues(SortedMap, Function) Maps.tran
Multimap*	Multimaps.transformValues(Multimap, Function) Multi
ListMultimap*	Multimaps.transformValues(ListMultimap, Function) M
Table	Tables.transformValues(Table, Function)

*Map和 Multimap 有特殊的方法，其中有个 EntryTransformer<K, V1, V2> 参数，它可以使用旧的键值来计算，并且用计算结果替换旧值。

*对Set的转换操作被省略了，因为不能有效支持contains(Object)操作——译者注：懒视图实际上不会全部计算转换后的***Set***元素，因此不能高效地支持***contains(Object)***。请改用Sets.newHashSet(Collections2.transform(set, function))进行拷贝转换。

```
List<String> names;
Map<String, Person> personWithName;
List<Person> people = Lists.transform(names, Functions.forMap(pe
rsonWithName));

ListMultimap<String, String> firstNameToLastNames;
// maps first names to all last names of people with that first
name

ListMultimap<String, String> firstNameToName = Multimaps.transfo
rmEntries(firstNameToLastNames,
    new EntryTransformer<String, String, String> () {
        public String transformEntry(String firstName, String la
stName) {
            return firstName + " " + lastName;
        }
    });
```

可以组合Function使用的类包括：

Ordering	**`Ordering.onResultOf(Function)`**
Predicate	`Predicates.compose(Predicate, Function)`
Equivalence	`Equivalence.onResultOf(Function)`
Supplier	`Suppliers.compose(Function, Supplier)`
Function	`Functions.compose(Function, Function)`

此外，ListenableFuture API支持转换ListenableFuture。Futures也提供了接受 `AsyncFunction` 参数的方法。AsyncFunction是Function的变种，它允许异步计算值。

`Futures.transform(ListenableFuture, Function)`
`Futures.transform(ListenableFuture, Function, Executor)`
`Futures.transform(ListenableFuture, AsyncFunction)`
`Futures.transform(ListenableFuture, AsyncFunction, Executor)`

5-并发

5.1-google Guava包的ListenableFuture解析

原文地址 译者：罗立树 校对：方腾飞

并发编程是一个难题，但是一个强大而简单的抽象可以显著的简化并发的编写。出于这样的考虑，Guava 定义了 ListenableFuture接口并继承了JDK concurrent包下的 `Future` 接口。

我们强烈地建议你在代码中多使用 **`ListenableFuture来代替JDK的`** **`Future`**，因为：

- `大多数Futures` 方法中需要它。
- 转到 `ListenableFuture` 编程比较容易。
- Guava提供的通用公共类封装了公共的操作方方法，不需要提供Future和 `ListenableFuture的扩展方法。`

接口

传统JDK中的Future通过异步的方式计算返回结果:在多线程运算中可能或者可能在没有结束返回结果，Future是运行中的多线程的一个引用句柄，确保在服务执行返回一个Result。

ListenableFuture可以允许你注册回调方法(callbacks)，在运算（多线程执行）完成的时候进行调用, 或者在运算（多线程执行）完成后立即执行。这样简单的改进，使得可以明显的支持更多的操作，这样的功能在JDK concurrent中的Future是不支持的。

`ListenableFuture` 中的基础方法是 `addListener(Runnable, Executor)`, 该方法会在多线程运算完的时候，指定的Runnable参数传入的对象会被指定的Executor执行。

添加回调（Callbacks）

多数用户喜欢使用 Futures.addCallback(ListenableFuture<V>, FutureCallback<V>, Executor)的方式, 或者 另外一个版本version（译者注：addCallback(ListenableFuture<V> future,FutureCallback<? super V> callback)），默认是采用 `MoreExecutors.sameThreadExecutor()线程池`, 为了简化使用，Callback采用轻量级的设计. `FutureCallback<V>` 中实现了两个方法:

- `onSuccess(V)`,在Future成功的时候执行，根据Future结果来判断。
- `onFailure(Throwable)`, 在Future失败的时候执行，根据Future结果来判断。

ListenableFuture的创建

对应JDK中的 `ExecutorService.submit(Callable)` 提交多线程异步运算的方式，Guava 提供了 `ListeningExecutorService` 接口, 该接口返回 `ListenableFuture` 而相应的 `ExecutorService` 返回普通的 `Future` 。将 `ExecutorService` 转为 `ListeningExecutorService`， 可以使用MoreExecutors.listeningDecorator(ExecutorService)进行装饰。

```
ListeningExecutorService service = MoreExecutors.listeningDecora
tor(Executors.newFixedThreadPool(10));
ListenableFuture explosion = service.submit(new Callable() {
  public Explosion call() {
    return pushBigRedButton();
  }
});
Futures.addCallback(explosion, new FutureCallback() {
  // we want this handler to run immediately after we push the b
ig red button!
  public void onSuccess(Explosion explosion) {
    walkAwayFrom(explosion);
  }
  public void onFailure(Throwable thrown) {
    battleArchNemesis(); // escaped the explosion!
  }
});
```

另外, 假如你是从 FutureTask转换而来的, Guava 提供 `ListenableFutureTask.create(Callable<V>)` 和 `ListenableFutureTask.create(Runnable, V)` . 和 JDK不同的是, `ListenableFutureTask` 不能随意被继承（译者注：ListenableFutureTask中的done方法实现了调用listener的操作）。

假如你喜欢抽象的方式来设置future的值，而不是想实现接口中的方法，可以考虑继承抽象类 `AbstractFuture<V>` 或者直接使用 `SettableFuture` 。

假如你必须将其他API提供的Future转换成 `ListenableFuture` ，你没有别的方法只能采用硬编码的方式 `JdkFutureAdapters.listenInPoolThread(Future)` 来将 `Future` 转换成 `ListenableFuture` 。尽可能地采用修改原生的代码返回 `ListenableFuture` 会更好一些。

Application

使用 `ListenableFuture` 最重要的理由是它可以进行一系列的复杂链式的异步操作。

```
ListenableFuture rowKeyFuture = indexService.lookUp(query);
AsyncFunction<RowKey, QueryResult> queryFunction =
new AsyncFunction<RowKey, QueryResult>() {
public ListenableFuture apply(RowKey rowKey) {
return dataService.read(rowKey);
}
};
ListenableFuture queryFuture = Futures.transform(rowKeyFuture, q
ueryFunction, queryExecutor);
```

其他更多的操作可以更加有效的支持而JDK中的Future是没法支持的.

不同的操作可以在不同的Executors中执行，单独的 `ListenableFuture` 可以有多个操作等待。

当一个操作开始的时候其他的一些操作也会尽快开始执行–“fan-out”– `ListenableFuture` 能够满足这样的场景：促发所有的回调（callbacks）。反之更简单的工作是，同样可以满足“fan-in”场景，促发 `ListenableFuture` 获取（get）计算结果，同时其它的Futures也会尽快执行：可以参考 the implementation of `Futures.allAsList` 。（译者注：fan-in和fan-out是软件设计的一个术语，可以参考这里： http://baike.baidu.com/view/388892.htm#1 或者看这里的解析 Design Principles: Fan-In vs Fan-Out ，这里fan-out的实现就是封装的ListenableFuture通过回调，调用其它代码片段。fan-in的意义是可以调用其它Future）

方法
transform(ListenableFuture<A>, AsyncFunction<A, B>, Ex
transform(ListenableFuture<A>, Function<A, B>, Executo
allAsList(Iterable<ListenableFuture<V>>)
successfulAsList(Iterable<ListenableFuture<V>>)

AsyncFunction<A, B> 中提供一个方法 ListenableFuture<B> apply(A input)，它可以被用于异步变换值。

```
List<ListenableFuture> queries;
// The queries go to all different data centers, but we want to
wait until they're all done or failed.

ListenableFuture<List> successfulQueries = Futures.successfulAsL
ist(queries);

Futures.addCallback(successfulQueries, callbackOnSuccessfulQueri
es);
```

CheckedFuture

Guava也提供了 CheckedFuture<V, X extends Exception> 接口。 CheckedFuture 是一个 ListenableFuture ，其中包含了多个版本的get 方法，方法声明抛出检查异常.这样使得创建一个在执行逻辑中可以抛出异常的Future更加容易 。将 ListenableFuture 转换成 CheckedFuture ，可以使用

Futures.makeChecked(ListenableFuture<V>, Function<Exceptio 。Guava也提供了 CheckedFuture<V, X extends Exception> 接口。 CheckedFuture 是一个 ListenableFuture ，其中包含了多个版本的get方法，方法声明抛出检查异常.这样使得创建一个在执行逻辑中可以抛出异常的Future更加容易 。将 ListenableFuture 转换成 CheckedFuture ，可以使用 Futures.makeChecked(ListenableFuture<V>, Function<Exceptio 。Guava也提供了 CheckedFuture<V, X extends Exception> 接口。 CheckedFuture 是一个 ListenableFuture ，其中包含了多个版本的get方法，方法声明抛出检查异常.这样使得创建一个在执行逻辑中可以抛出异常的Future更加容易 。将 ListenableFuture 转换成 CheckedFuture ，可以使用 Futures.makeChecked(ListenableFuture<V>, Function<Exceptio 。Guava也提供了 CheckedFuture<V, X extends Exception> 接口。 CheckedFuture 是一个 ListenableFuture ，其中包含了多个版本的get方法，方法声明抛出检查异常.这样使得创建一个在执行逻辑中可以抛出异常的Future更加容易 。将 ListenableFuture 转换成 CheckedFuture ，可以使用 Futures.makeChecked(ListenableFuture<V>, Function<Exceptio 。Guava也提供了 CheckedFuture<V, X extends Exception> 接口。 CheckedFuture 是一个 ListenableFuture ，其中包含了多个版本的get方法，方法声明抛出检查异常.这样使得创建一个在执行逻辑中可以抛出异常的Future更加容易 。将 ListenableFuture 转换成 CheckedFuture ，可以使用 Futures.makeChecked(ListenableFuture<V>, Function<Exceptio 。Guava也提供了 CheckedFuture<V, X extends Exception> 接口。 CheckedFuture 是一个 ListenableFuture ，其中包含了多个版本的get方法，方法声明抛出检查异常.这样使得创建一个在执行逻辑中可以抛出异常的Future更加容易 。将 ListenableFuture 转换成 CheckedFuture ，可以使用 Futures.makeChecked(ListenableFuture<V>, Function<Exceptio 。Guava也提供了 CheckedFuture<V, X extends Exception> 接口。 CheckedFuture 是一个 ListenableFuture ，其中包含了多个版本的get方法，方法声明抛出检查异常.这样使得创建一个在执行逻辑中可以抛出异常的Future更加容易 。将 ListenableFuture 转换成 CheckedFuture ，可以使用 Futures.makeChecked(ListenableFuture<V>, Function<Exceptio 。

5.2-Google-Guava Concurrent包里的Service框架浅析

原文地址 译文地址 译者：何一昕 校对：方腾飞

概述

Guava包里的Service接口用于封装一个服务对象的运行状态、包括start和stop等方法。例如web服务器，RPC服务器、计时器等可以实现这个接口。对此类服务的状态管理并不轻松、需要对服务的开启/关闭进行妥善管理、特别是在多线程环境下尤为复杂。Guava包提供了一些基础类帮助你管理复杂的状态转换逻辑和同步细节。

使用一个服务

一个服务正常生命周期有：

- Service.State.NEW
- Service.State.STARTING
- Service.State.RUNNING
- Service.State.STOPPING
- Service.State.TERMINATED

服务一旦被停止就无法再重新启动了。如果服务在starting、running、stopping状态出现问题、会进入 `Service.State.FAILED` .状态。调用 `startAsync()` 方法可以异步开启一个服务,同时返回this对象形成方法调用链。注意：只有在当前服务的状态是 `NEW` 时才能调用startAsync()方法，因此最好在应用中有一个统一的地方初始化相关服务。停止一个服务也是类似的、使用异步方法 `stopAsync()` 。但是不像startAsync(),多次调用这个方法是安全的。这是为了方便处理关闭服务时候的锁竞争问题。

Service也提供了一些方法用于等待服务状态转换的完成:

通过 `addListener()` 方法异步添加监听器。此方法允许你添加一个 `Service.Listener` 、它会在每次服务状态转换的时候被调用。注意：最好在服务启动之前添加Listener（这时的状态是NEW）、否则之前已发生的状态转换事件是无法在新添加的Listener上被重新触发的。

同步使用 `awaitRunning()` 。这个方法不能被打断、不强制捕获异常、一旦服务启动就会返回。如果服务没有成功启动，会抛出IllegalStateException异常。同样的， `awaitTerminated()` 方法会等待服务达到终止状态（ `TERMINATED` 或者 `FAILED` ）。两个方法都有重载方法允许传入超时时间。

`Service` 接口本身实现起来会比较复杂、且容易碰到一些捉摸不透的问题。因此我们不推荐直接实现这个接口。而是请继承Guava包里已经封装好的基础抽象类。每个基础类支持一种特定的线程模型。

基础实现类

AbstractIdleService

`AbstractIdleService` 类简单实现了Service接口、其在running状态时不会执行任何动作–因此在running时也不需要启动线程–但需要处理开启/关闭动作。要实现一个此类的服务，只需继承AbstractIdleService类，然后自己实现 `startUp()` 和 `shutDown()` 方法就可以了。

```
protected void startUp() {
servlets.add(new GcStatsServlet());
}
protected void shutDown() {}
```

如上面的例子、由于任何请求到GcStatsServlet时已经会有现成线程处理了，所以在服务运行时就不需要做什么额外动作了。

AbstractExecutionThreadService

`AbstractExecutionThreadService` 通过单线程处理启动、运行、和关闭等操作。你必须重载run()方法，同时需要能响应停止服务的请求。具体的实现可以在一个循环内做处理：

```
 public void run() {
   while (isRunning()) {
     // perform a unit of work
   }
 }
```

另外，你还可以重载 `triggerShutdown()` 方法让run()方法结束返回。

重载startUp()和shutDown()方法是可选的，不影响服务本身状态的管理

```
 protected void startUp() {
dispatcher.listenForConnections(port, queue);
 }
 protected void run() {
   Connection connection;
   while ((connection = queue.take() != POISON)) {
     process(connection);
   }
 }
 protected void triggerShutdown() {
   dispatcher.stopListeningForConnections(queue);
   queue.put(POISON);
 }
```

start()内部会调用startUp()方法，创建一个线程、然后在线程内调用run()方法。stop()会调用 triggerShutdown()方法并且等待线程终止。

AbstractScheduledService

`AbstractScheduledService` 类用于在运行时处理一些周期性的任务。子类可以实现 `runOneIteration()` 方法定义一个周期执行的任务，以及相应的startUp()和shutDown()方法。为了能够描述执行周期，你需要实现 `scheduler()` 方法。通常情况下，你可以使用 `AbstractScheduledService.Scheduler` 类提供的两种调度器： `newFixedRateSchedule(initialDelay, delay, TimeUnit)` 和 `newFixedDelaySchedule(initialDelay, delay, TimeUnit)` ，类似于JDK并发包中ScheduledExecutorService类提供的两种调度方式。如要自定义schedules则可以使用 `CustomScheduler` 类来辅助实现；具体用法见javadoc。

AbstractService

如需要自定义的线程管理、可以通过扩展 `AbstractService` 类来实现。一般情况下、使用上面的几个实现类就已经满足需求了，但如果在服务执行过程中有一些特定的线程处理需求、则建议继承AbstractService类。

继承AbstractService方法必须实现两个方法.

- **`doStart()`**：首次调用startAsync()时会同时调用doStart(),doStart()内部需要处理所有的初始化工作、如果启动成功则调用 `notifyStarted()` 方法；启动失败则调用 `notifyFailed()`
- **`doStop()`**：首次调用stopAsync()会同时调用doStop(),doStop()要做的事情就是停止服务，如果停止成功则调用 `notifyStopped()` 方法；停止失败则调用 `notifyFailed()` 方法。

doStart和doStop方法的实现需要考虑下性能，尽可能的低延迟。如果初始化的开销较大，如读文件，打开网络连接，或者其他任何可能引起阻塞的操作，建议移到另外一个单独的线程去处理。

使用**ServiceManager**

除了对Service接口提供基础的实现类，Guava还提供了 `ServiceManager` 类使得涉及到多个Service集合的操作更加容易。通过实例化ServiceManager类来创建一个Service集合，你可以通过以下方法来管理它们：

- **`startAsync()`** ： 将启动所有被管理的服务。如果当前服务的状态都是NEW的话、那么你只能调用该方法一次、这跟 Service#startAsync()是一样的。
- **`stopAsync()`** ：将停止所有被管理的服务。
- **`addListener`** ：会添加一个 `ServiceManager.Listener` ，在服务状态转换中会调用该Listener
- **`awaitHealthy()`** ：会等待所有的服务达到Running状态
- **`awaitStopped()`** ：会等待所有服务达到终止状态

检测类的方法有：

- **`isHealthy()`** **：**如果所有的服务处于Running状态、会返回True
- **`servicesByState()`** ：以状态为索引返回当前所有服务的快照
- **`startupTimes()`** ：返回一个Map对象，记录被管理的服务启动的耗时、以毫秒为单位，同时Map默认按启动时间排序。

我们建议整个服务的生命周期都能通过ServiceManager来管理，不过即使状态转换是通过其他机制触发的、也不影响ServiceManager方法的正确执行。例如：当一个服务不是通过startAsync()、而是其他机制启动时，listeners 仍然可以被正常调用、awaitHealthy()也能够正常工作。ServiceManager 唯一强制的要求是当其被创建时所有的服务必须处于New状态。

附：TestCase、也可以作为练习Demo

ServiceTest

```
</pre>
/*
 * Copyright (C) 2013 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License"
);
 * you may not use this file except in compliance with the Licen
se.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, so
ftware
 * distributed under the License is distributed on an "AS IS" BA
SIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
 * See the License for the specific language governing permissio
ns and
 * limitations under the License.
 */

package com.google.common.util.concurrent;

import static com.google.common.util.concurrent.Service.State.FA
ILED;
import static com.google.common.util.concurrent.Service.State.NE
W;
import static com.google.common.util.concurrent.Service.State.RU
NNING;
import static com.google.common.util.concurrent.Service.State.ST
ARTING;
import static com.google.common.util.concurrent.Service.State.ST
OPPING;
import static com.google.common.util.concurrent.Service.State.TE
RMINATED;

import junit.framework.TestCase;

/**
 * Unit tests for {@link Service}
```

```
 */
public class ServiceTest extends TestCase {

/** Assert on the comparison ordering of the State enum since we
 guarantee it. */
 public void testStateOrdering() {
 // List every valid (direct) state transition.
 assertLessThan(NEW, STARTING);
 assertLessThan(NEW, TERMINATED);

 assertLessThan(STARTING, RUNNING);
 assertLessThan(STARTING, STOPPING);
 assertLessThan(STARTING, FAILED);

 assertLessThan(RUNNING, STOPPING);
 assertLessThan(RUNNING, FAILED);

 assertLessThan(STOPPING, FAILED);
 assertLessThan(STOPPING, TERMINATED);
 }

 private static <T extends Comparable<? super T>> void assertLes
sThan(T a, T b) {
 if (a.compareTo(b) >= 0) {
 fail(String.format("Expected %s to be less than %s", a, b));
 }
 }
}
<pre>
```

AbstractIdleServiceTest

```
/*
 * Copyright (C) 2009 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License"
);
 * you may not use this file except in compliance with the Licen
se.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, so
ftware
 * distributed under the License is distributed on an "AS IS" BA
SIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
 * See the License for the specific language governing permissio
ns and
```

```
 * limitations under the License.
 */

package com.google.common.util.concurrent;

import static org.truth0.Truth.ASSERT;

import com.google.common.collect.Lists;

import junit.framework.TestCase;

import java.util.List;
import java.util.concurrent.Executor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

/**
 * Tests for {@link AbstractIdleService}.
 *
 * @author Chris Nokleberg
 * @author Ben Yu
 */
public class AbstractIdleServiceTest extends TestCase {

// Functional tests using real thread. We only verify publicly v
isible state.
 // Interaction assertions are done by the single-threaded unit
tests.

public static class FunctionalTest extends TestCase {

private static class DefaultService extends AbstractIdleService
{
 @Override protected void startUp() throws Exception {}
 @Override protected void shutDown() throws Exception {}
 }

public void testServiceStartStop() throws Exception {
 AbstractIdleService service = new DefaultService();
 service.startAsync().awaitRunning();
 assertEquals(Service.State.RUNNING, service.state());
 service.stopAsync().awaitTerminated();
 assertEquals(Service.State.TERMINATED, service.state());
 }

public void testStart_failed() throws Exception {
 final Exception exception = new Exception("deliberate");
 AbstractIdleService service = new DefaultService() {
 @Override protected void startUp() throws Exception {
 throw exception;
 }
 };
 try {
```

```
 service.startAsync().awaitRunning();
 fail();
 } catch (RuntimeException e) {
 assertSame(exception, e.getCause());
 }
 assertEquals(Service.State.FAILED, service.state());
 }

public void testStop_failed() throws Exception {
 final Exception exception = new Exception("deliberate");
 AbstractIdleService service = new DefaultService() {
 @Override protected void shutDown() throws Exception {
 throw exception;
 }
 };
 service.startAsync().awaitRunning();
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (RuntimeException e) {
 assertSame(exception, e.getCause());
 }
 assertEquals(Service.State.FAILED, service.state());
 }
 }

public void testStart() {
 TestService service = new TestService();
 assertEquals(0, service.startUpCalled);
 service.startAsync().awaitRunning();
 assertEquals(1, service.startUpCalled);
 assertEquals(Service.State.RUNNING, service.state());
 ASSERT.that(service.transitionStates).has().exactly(Service.Sta
te.STARTING).inOrder();
 }

public void testStart_failed() {
 final Exception exception = new Exception("deliberate");
 TestService service = new TestService() {
 @Override protected void startUp() throws Exception {
 super.startUp();
 throw exception;
 }
 };
 assertEquals(0, service.startUpCalled);
 try {
 service.startAsync().awaitRunning();
 fail();
 } catch (RuntimeException e) {
 assertSame(exception, e.getCause());
 }
 assertEquals(1, service.startUpCalled);
 assertEquals(Service.State.FAILED, service.state());
```

```
 ASSERT.that(service.transitionStates).has().exactly(Service.Sta
te.STARTING).inOrder();
 }

public void testStop_withoutStart() {
 TestService service = new TestService();
 service.stopAsync().awaitTerminated();
 assertEquals(0, service.startUpCalled);
 assertEquals(0, service.shutDownCalled);
 assertEquals(Service.State.TERMINATED, service.state());
 ASSERT.that(service.transitionStates).isEmpty();
 }

public void testStop_afterStart() {
 TestService service = new TestService();
 service.startAsync().awaitRunning();
 assertEquals(1, service.startUpCalled);
 assertEquals(0, service.shutDownCalled);
 service.stopAsync().awaitTerminated();
 assertEquals(1, service.startUpCalled);
 assertEquals(1, service.shutDownCalled);
 assertEquals(Service.State.TERMINATED, service.state());
 ASSERT.that(service.transitionStates)
 .has().exactly(Service.State.STARTING, Service.State.STOPPING).
inOrder();
 }

public void testStop_failed() {
 final Exception exception = new Exception("deliberate");
 TestService service = new TestService() {
 @Override protected void shutDown() throws Exception {
 super.shutDown();
 throw exception;
 }
 };
 service.startAsync().awaitRunning();
 assertEquals(1, service.startUpCalled);
 assertEquals(0, service.shutDownCalled);
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (RuntimeException e) {
 assertSame(exception, e.getCause());
 }
 assertEquals(1, service.startUpCalled);
 assertEquals(1, service.shutDownCalled);
 assertEquals(Service.State.FAILED, service.state());
 ASSERT.that(service.transitionStates)
 .has().exactly(Service.State.STARTING, Service.State.STOPPING).
inOrder();
 }

public void testServiceToString() {
```

```
 AbstractIdleService service = new TestService();
 assertEquals("TestService [NEW]", service.toString());
 service.startAsync().awaitRunning();
 assertEquals("TestService [RUNNING]", service.toString());
 service.stopAsync().awaitTerminated();
 assertEquals("TestService [TERMINATED]", service.toString());
 }

public void testTimeout() throws Exception {
 // Create a service whose executor will never run its commands
 Service service = new TestService() {
 @Override protected Executor executor() {
 return new Executor() {
 @Override public void execute(Runnable command) {}
 };
 }
 };
 try {
 service.startAsync().awaitRunning(1, TimeUnit.MILLISECONDS);
 fail("Expected timeout");
 } catch (TimeoutException e) {
 ASSERT.that(e.getMessage()).contains(Service.State.STARTING.toS
tring());
 }
 }

private static class TestService extends AbstractIdleService {
 int startUpCalled = 0;
 int shutDownCalled = 0;
 final List<State> transitionStates = Lists.newArrayList();

@Override protected void startUp() throws Exception {
 assertEquals(0, startUpCalled);
 assertEquals(0, shutDownCalled);
 startUpCalled++;
 assertEquals(State.STARTING, state());
 }

@Override protected void shutDown() throws Exception {
 assertEquals(1, startUpCalled);
 assertEquals(0, shutDownCalled);
 shutDownCalled++;
 assertEquals(State.STOPPING, state());
 }

@Override protected Executor executor() {
 transitionStates.add(state());
 return MoreExecutors.sameThreadExecutor();
 }
 }
}

<pre>
```

AbstractScheduledServiceTest

```
</pre>
/*
 * Copyright (C) 2011 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License"
);
 * you may not use this file except in compliance with the Licen
se.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, so
ftware
 * distributed under the License is distributed on an "AS IS" BA
SIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
 * See the License for the specific language governing permissio
ns and
 * limitations under the License.
 */

package com.google.common.util.concurrent;

import com.google.common.util.concurrent.AbstractScheduledServic
e.Scheduler;
import com.google.common.util.concurrent.Service.State;

import junit.framework.TestCase;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Unit test for {@link AbstractScheduledService}.
 *
 * @author Luke Sandberg
 */
```

```
public class AbstractScheduledServiceTest extends TestCase {

volatile Scheduler configuration = Scheduler.newFixedDelaySchedu
le(0, 10, TimeUnit.MILLISECONDS);
 volatile ScheduledFuture<?> future = null;

volatile boolean atFixedRateCalled = false;
 volatile boolean withFixedDelayCalled = false;
 volatile boolean scheduleCalled = false;

final ScheduledExecutorService executor = new ScheduledThreadPoo
lExecutor(10) {
 @Override
 public ScheduledFuture<?> scheduleWithFixedDelay(Runnable comma
nd, long initialDelay,
 long delay, TimeUnit unit) {
 return future = super.scheduleWithFixedDelay(command, initialDe
lay, delay, unit);
 }
 };

public void testServiceStartStop() throws Exception {
 NullService service = new NullService();
 service.startAsync().awaitRunning();
 assertFalse(future.isDone());
 service.stopAsync().awaitTerminated();
 assertTrue(future.isCancelled());
 }

private class NullService extends AbstractScheduledService {
 @Override protected void runOneIteration() throws Exception {}
 @Override protected Scheduler scheduler() { return configuratio
n; }
 @Override protected ScheduledExecutorService executor() { retur
n executor; }
 }

public void testFailOnExceptionFromRun() throws Exception {
 TestService service = new TestService();
 service.runException = new Exception();
 service.startAsync().awaitRunning();
 service.runFirstBarrier.await();
 service.runSecondBarrier.await();
 try {
 future.get();
 fail();
 } catch (ExecutionException e) {
 // An execution exception holds a runtime exception (from throw
ables.propogate) that holds our
 // original exception.
 assertEquals(service.runException, e.getCause().getCause());
 }
 assertEquals(service.state(), Service.State.FAILED);
```

```
 }

public void testFailOnExceptionFromStartUp() {
 TestService service = new TestService();
 service.startUpException = new Exception();
 try {
 service.startAsync().awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.startUpException, e.getCause());
 }
 assertEquals(0, service.numberOfTimesRunCalled.get());
 assertEquals(Service.State.FAILED, service.state());
 }

public void testFailOnExceptionFromShutDown() throws Exception {
 TestService service = new TestService();
 service.shutDownException = new Exception();
 service.startAsync().awaitRunning();
 service.runFirstBarrier.await();
 service.stopAsync();
 service.runSecondBarrier.await();
 try {
 service.awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.shutDownException, e.getCause());
 }
 assertEquals(Service.State.FAILED, service.state());
 }

public void testRunOneIterationCalledMultipleTimes() throws Exce
ption {
 TestService service = new TestService();
 service.startAsync().awaitRunning();
 for (int i = 1; i < 10; i++) {
 service.runFirstBarrier.await();
 assertEquals(i, service.numberOfTimesRunCalled.get());
 service.runSecondBarrier.await();
 }
 service.runFirstBarrier.await();
 service.stopAsync();
 service.runSecondBarrier.await();
 service.stopAsync().awaitTerminated();
 }

public void testExecutorOnlyCalledOnce() throws Exception {
 TestService service = new TestService();
 service.startAsync().awaitRunning();
 // It should be called once during startup.
 assertEquals(1, service.numberOfTimesExecutorCalled.get());
 for (int i = 1; i < 10; i++) {
 service.runFirstBarrier.await();
```

```
 assertEquals(i, service.numberOfTimesRunCalled.get());
 service.runSecondBarrier.await();
 }
 service.runFirstBarrier.await();
 service.stopAsync();
 service.runSecondBarrier.await();
 service.stopAsync().awaitTerminated();
 // Only called once overall.
 assertEquals(1, service.numberOfTimesExecutorCalled.get());
 }

public void testDefaultExecutorIsShutdownWhenServiceIsStopped()
throws Exception {
 final CountDownLatch terminationLatch = new CountDownLatch(1);
 AbstractScheduledService service = new AbstractScheduledService
() {
 volatile ScheduledExecutorService executorService;
 @Override protected void runOneIteration() throws Exception {}

@Override protected ScheduledExecutorService executor() {
 if (executorService == null) {
 executorService = super.executor();
 // Add a listener that will be executed after the listener that
 shuts down the executor.
 addListener(new Listener() {
 @Override public void terminated(State from) {
 terminationLatch.countDown();
 }
 }, MoreExecutors.sameThreadExecutor());
 }
 return executorService;
 }

@Override protected Scheduler scheduler() {
 return Scheduler.newFixedDelaySchedule(0, 1, TimeUnit.MILLISECO
NDS);
 }};

service.startAsync();
 assertFalse(service.executor().isShutdown());
 service.awaitRunning();
 service.stopAsync();
 terminationLatch.await();
 assertTrue(service.executor().isShutdown());
 assertTrue(service.executor().awaitTermination(100, TimeUnit.MI
LLISECONDS));
 }

public void testDefaultExecutorIsShutdownWhenServiceFails() thro
ws Exception {
 final CountDownLatch failureLatch = new CountDownLatch(1);
 AbstractScheduledService service = new AbstractScheduledService
() {
```

```
 volatile ScheduledExecutorService executorService;
 @Override protected void runOneIteration() throws Exception {}

@Override protected void startUp() throws Exception {
 throw new Exception("Failed");
 }

@Override protected ScheduledExecutorService executor() {
 if (executorService == null) {
 executorService = super.executor();
 // Add a listener that will be executed after the listener that
 shuts down the executor.
 addListener(new Listener() {
 @Override public void failed(State from, Throwable failure) {
 failureLatch.countDown();
 }
 }, MoreExecutors.sameThreadExecutor());
 }
 return executorService;
 }

@Override protected Scheduler scheduler() {
 return Scheduler.newFixedDelaySchedule(0, 1, TimeUnit.MILLISECO
NDS);
 }};

try {
 service.startAsync().awaitRunning();
 fail("Expected service to fail during startup");
 } catch (IllegalStateException expected) {}
 failureLatch.await();
 assertTrue(service.executor().isShutdown());
 assertTrue(service.executor().awaitTermination(100, TimeUnit.MI
LLISECONDS));
 }

public void testSchedulerOnlyCalledOnce() throws Exception {
 TestService service = new TestService();
 service.startAsync().awaitRunning();
 // It should be called once during startup.
 assertEquals(1, service.numberOfTimesSchedulerCalled.get());
 for (int i = 1; i < 10; i++) {
 service.runFirstBarrier.await();
 assertEquals(i, service.numberOfTimesRunCalled.get());
 service.runSecondBarrier.await();
 }
 service.runFirstBarrier.await();
 service.stopAsync();
 service.runSecondBarrier.await();
 service.awaitTerminated();
 // Only called once overall.
 assertEquals(1, service.numberOfTimesSchedulerCalled.get());
 }
```

```
private class TestService extends AbstractScheduledService {
 CyclicBarrier runFirstBarrier = new CyclicBarrier(2);
 CyclicBarrier runSecondBarrier = new CyclicBarrier(2);

volatile boolean startUpCalled = false;
 volatile boolean shutDownCalled = false;
 AtomicInteger numberOfTimesRunCalled = new AtomicInteger(0);
 AtomicInteger numberOfTimesExecutorCalled = new AtomicInteger(0
);
 AtomicInteger numberOfTimesSchedulerCalled = new AtomicInteger(
0);
 volatile Exception runException = null;
 volatile Exception startUpException = null;
 volatile Exception shutDownException = null;

@Override
 protected void runOneIteration() throws Exception {
 assertTrue(startUpCalled);
 assertFalse(shutDownCalled);
 numberOfTimesRunCalled.incrementAndGet();
 assertEquals(State.RUNNING, state());
 runFirstBarrier.await();
 runSecondBarrier.await();
 if (runException != null) {
 throw runException;
 }
 }

@Override
 protected void startUp() throws Exception {
 assertFalse(startUpCalled);
 assertFalse(shutDownCalled);
 startUpCalled = true;
 assertEquals(State.STARTING, state());
 if (startUpException != null) {
 throw startUpException;
 }
 }

@Override
 protected void shutDown() throws Exception {
 assertTrue(startUpCalled);
 assertFalse(shutDownCalled);
 shutDownCalled = true;
 if (shutDownException != null) {
 throw shutDownException;
 }
 }

@Override
 protected ScheduledExecutorService executor() {
 numberOfTimesExecutorCalled.incrementAndGet();
```

```
 return executor;
 }

@Override
 protected Scheduler scheduler() {
 numberOfTimesSchedulerCalled.incrementAndGet();
 return configuration;
 }
 }

public static class SchedulerTest extends TestCase {
 // These constants are arbitrary and just used to make sure tha
t the correct method is called
 // with the correct parameters.
 private static final int initialDelay = 10;
 private static final int delay = 20;
 private static final TimeUnit unit = TimeUnit.MILLISECONDS;

// Unique runnable object used for comparison.
 final Runnable testRunnable = new Runnable() {@Override public
void run() {}};
 boolean called = false;

private void assertSingleCallWithCorrectParameters(Runnable comm
and, long initialDelay,
 long delay, TimeUnit unit) {
 assertFalse(called); // only called once.
 called = true;
 assertEquals(SchedulerTest.initialDelay, initialDelay);
 assertEquals(SchedulerTest.delay, delay);
 assertEquals(SchedulerTest.unit, unit);
 assertEquals(testRunnable, command);
 }

public void testFixedRateSchedule() {
 Scheduler schedule = Scheduler.newFixedRateSchedule(initialDela
y, delay, unit);
 schedule.schedule(null, new ScheduledThreadPoolExecutor(1) {
 @Override
 public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
 long initialDelay,
 long period, TimeUnit unit) {
 assertSingleCallWithCorrectParameters(command, initialDelay, de
lay, unit);
 return null;
 }
 }, testRunnable);
 assertTrue(called);
 }

public void testFixedDelaySchedule() {
 Scheduler schedule = Scheduler.newFixedDelaySchedule(initialDel
ay, delay, unit);
```

```
 schedule.schedule(null, new ScheduledThreadPoolExecutor(10) {
 @Override
 public ScheduledFuture<?> scheduleWithFixedDelay(Runnable comma
nd, long initialDelay,
 long delay, TimeUnit unit) {
 assertSingleCallWithCorrectParameters(command, initialDelay, de
lay, unit);
 return null;
 }
 }, testRunnable);
 assertTrue(called);
 }

private class TestCustomScheduler extends AbstractScheduledServi
ce.CustomScheduler {
 public AtomicInteger scheduleCounter = new AtomicInteger(0);
 @Override
 protected Schedule getNextSchedule() throws Exception {
 scheduleCounter.incrementAndGet();
 return new Schedule(0, TimeUnit.SECONDS);
 }
 }

public void testCustomSchedule_startStop() throws Exception {
 final CyclicBarrier firstBarrier = new CyclicBarrier(2);
 final CyclicBarrier secondBarrier = new CyclicBarrier(2);
 final AtomicBoolean shouldWait = new AtomicBoolean(true);
 Runnable task = new Runnable() {
 @Override public void run() {
 try {
 if (shouldWait.get()) {
 firstBarrier.await();
 secondBarrier.await();
 }
 } catch (Exception e) {
 throw new RuntimeException(e);
 }
 }
 };
 TestCustomScheduler scheduler = new TestCustomScheduler();
 Future<?> future = scheduler.schedule(null, Executors.newSchedu
ledThreadPool(10), task);
 firstBarrier.await();
 assertEquals(1, scheduler.scheduleCounter.get());
 secondBarrier.await();
 firstBarrier.await();
 assertEquals(2, scheduler.scheduleCounter.get());
 shouldWait.set(false);
 secondBarrier.await();
 future.cancel(false);
 }

public void testCustomSchedulerServiceStop() throws Exception {
```

```
 TestAbstractScheduledCustomService service = new TestAbstractSc
heduledCustomService();
 service.startAsync().awaitRunning();
 service.firstBarrier.await();
 assertEquals(1, service.numIterations.get());
 service.stopAsync();
 service.secondBarrier.await();
 service.awaitTerminated();
 // Sleep for a while just to ensure that our task wasn't called
 again.
 Thread.sleep(unit.toMillis(3 * delay));
 assertEquals(1, service.numIterations.get());
 }

public void testBig() throws Exception {
 TestAbstractScheduledCustomService service = new TestAbstractSc
heduledCustomService() {
 @Override protected Scheduler scheduler() {
 return new AbstractScheduledService.CustomScheduler() {
 @Override
 protected Schedule getNextSchedule() throws Exception {
 // Explicitly yield to increase the probability of a pathologic
al scheduling.
 Thread.yield();
 return new Schedule(0, TimeUnit.SECONDS);
 }
 };
 }
 };
 service.useBarriers = false;
 service.startAsync().awaitRunning();
 Thread.sleep(50);
 service.useBarriers = true;
 service.firstBarrier.await();
 int numIterations = service.numIterations.get();
 service.stopAsync();
 service.secondBarrier.await();
 service.awaitTerminated();
 assertEquals(numIterations, service.numIterations.get());
 }

private static class TestAbstractScheduledCustomService extends
AbstractScheduledService {
 final AtomicInteger numIterations = new AtomicInteger(0);
 volatile boolean useBarriers = true;
 final CyclicBarrier firstBarrier = new CyclicBarrier(2);
 final CyclicBarrier secondBarrier = new CyclicBarrier(2);

@Override protected void runOneIteration() throws Exception {
 numIterations.incrementAndGet();
 if (useBarriers) {
 firstBarrier.await();
 secondBarrier.await();
```

```
 }
 }

@Override protected ScheduledExecutorService executor() {
 // use a bunch of threads so that weird overlapping schedules a
re more likely to happen.
 return Executors.newScheduledThreadPool(10);
 }

@Override protected void startUp() throws Exception {}

@Override protected void shutDown() throws Exception {}

@Override protected Scheduler scheduler() {
 return new CustomScheduler() {
 @Override
 protected Schedule getNextSchedule() throws Exception {
 return new Schedule(delay, unit);
 }};
 }
 }

public void testCustomSchedulerFailure() throws Exception {
 TestFailingCustomScheduledService service = new TestFailingCust
omScheduledService();
 service.startAsync().awaitRunning();
 for (int i = 1; i < 4; i++) {
 service.firstBarrier.await();
 assertEquals(i, service.numIterations.get());
 service.secondBarrier.await();
 }
 Thread.sleep(1000);
 try {
 service.stopAsync().awaitTerminated(100, TimeUnit.SECONDS);
 fail();
 } catch (IllegalStateException e) {
 assertEquals(State.FAILED, service.state());
 }
 }

private static class TestFailingCustomScheduledService extends A
bstractScheduledService {
 final AtomicInteger numIterations = new AtomicInteger(0);
 final CyclicBarrier firstBarrier = new CyclicBarrier(2);
 final CyclicBarrier secondBarrier = new CyclicBarrier(2);

@Override protected void runOneIteration() throws Exception {
 numIterations.incrementAndGet();
 firstBarrier.await();
 secondBarrier.await();
 }

@Override protected ScheduledExecutorService executor() {
```

```
 // use a bunch of threads so that weird overlapping schedules a
re more likely to happen.
 return Executors.newScheduledThreadPool(10);
 }

@Override protected Scheduler scheduler() {
 return new CustomScheduler() {
 @Override
 protected Schedule getNextSchedule() throws Exception {
 if (numIterations.get() > 2) {
 throw new IllegalStateException("Failed");
 }
 return new Schedule(delay, unit);
 }};
 }
 }
 }
}
<pre>
```

AbstractServiceTest

```
</pre>
/*
 * Copyright (C) 2009 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License"
);
 * you may not use this file except in compliance with the Licen
se.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, so
ftware
 * distributed under the License is distributed on an "AS IS" BA
SIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
 * See the License for the specific language governing permissio
ns and
 * limitations under the License.
 */

package com.google.common.util.concurrent;

import static java.lang.Thread.currentThread;
import static java.util.concurrent.TimeUnit.SECONDS;

import com.google.common.collect.ImmutableList;
```

```
import com.google.common.collect.Iterables;
import com.google.common.collect.Lists;
import com.google.common.util.concurrent.Service.Listener;
import com.google.common.util.concurrent.Service.State;

import junit.framework.TestCase;

import java.lang.Thread.UncaughtExceptionHandler;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicReference;

import javax.annotation.concurrent.GuardedBy;

/**
 * Unit test for {@link AbstractService}.
 *
 * @author Jesse Wilson
 */
public class AbstractServiceTest extends TestCase {

private Thread executionThread;
 private Throwable thrownByExecutionThread;

public void testNoOpServiceStartStop() throws Exception {
 NoOpService service = new NoOpService();
 RecordingListener listener = RecordingListener.record(service);

assertEquals(State.NEW, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.running);

service.startAsync();
 assertEquals(State.RUNNING, service.state());
 assertTrue(service.isRunning());
 assertTrue(service.running);

service.stopAsync();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.running);
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());
 }

public void testNoOpServiceStartAndWaitStopAndWait() throws Exce
```

```
ption {
 NoOpService service = new NoOpService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());
 }

public void testNoOpServiceStartAsyncAndAwaitStopAsyncAndAwait()
 throws Exception {
 NoOpService service = new NoOpService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());
 }

public void testNoOpServiceStopIdempotence() throws Exception {
 NoOpService service = new NoOpService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync();
 service.stopAsync();
 assertEquals(State.TERMINATED, service.state());
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());
 }

public void testNoOpServiceStopIdempotenceAfterWait() throws Exc
eption {
 NoOpService service = new NoOpService();

service.startAsync().awaitRunning();

service.stopAsync().awaitTerminated();
 service.stopAsync();
 assertEquals(State.TERMINATED, service.state());
 }

public void testNoOpServiceStopIdempotenceDoubleWait() throws Ex
ception {
 NoOpService service = new NoOpService();
```

```
service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync().awaitTerminated();
 service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());
 }

public void testNoOpServiceStartStopAndWaitUninterruptible()
 throws Exception {
 NoOpService service = new NoOpService();

currentThread().interrupt();
 try {
 service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());

assertTrue(currentThread().isInterrupted());
 } finally {
 Thread.interrupted(); // clear interrupt for future tests
 }
 }

private static class NoOpService extends AbstractService {
 boolean running = false;

@Override protected void doStart() {
 assertFalse(running);
 running = true;
 notifyStarted();
 }

@Override protected void doStop() {
 assertTrue(running);
 running = false;
 notifyStopped();
 }
 }

public void testManualServiceStartStop() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 assertEquals(State.STARTING, service.state());
 assertFalse(service.isRunning());
 assertTrue(service.doStartCalled);

service.notifyStarted(); // usually this would be invoked by ano
```

```
ther thread
 assertEquals(State.RUNNING, service.state());
 assertTrue(service.isRunning());

service.stopAsync();
 assertEquals(State.STOPPING, service.state());
 assertFalse(service.isRunning());
 assertTrue(service.doStopCalled);

service.notifyStopped(); // usually this would be invoked by ano
ther thread
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());

}

public void testManualServiceNotifyStoppedWhileRunning() throws
Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 service.notifyStarted();
 service.notifyStopped();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.doStopCalled);

assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.TERMINATED),
 listener.getStateHistory());
 }

public void testManualServiceStopWhileStarting() throws Exceptio
n {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 assertEquals(State.STARTING, service.state());
 assertFalse(service.isRunning());
 assertTrue(service.doStartCalled);
```

```
service.stopAsync();
 assertEquals(State.STOPPING, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.doStopCalled);

service.notifyStarted();
 assertEquals(State.STOPPING, service.state());
 assertFalse(service.isRunning());
 assertTrue(service.doStopCalled);

service.notifyStopped();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());
 }

/**
 * This tests for a bug where if {@link Service#stopAsync()} was
 called while the service was
 * {@link State#STARTING} more than once, the {@link Listener#st
opping(State)} callback would get
 * called multiple times.
 */
 public void testManualServiceStopMultipleTimesWhileStarting() t
hrows Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 final AtomicInteger stopppingCount = new AtomicInteger();
 service.addListener(new Listener() {
 @Override public void stopping(State from) {
 stopppingCount.incrementAndGet();
 }
 }, MoreExecutors.sameThreadExecutor());

service.startAsync();
 service.stopAsync();
 assertEquals(1, stopppingCount.get());
 service.stopAsync();
 assertEquals(1, stopppingCount.get());
 }

public void testManualServiceStopWhileNew() throws Exception {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);

service.stopAsync();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
 assertFalse(service.doStartCalled);
```

```
 assertFalse(service.doStopCalled);
 assertEquals(ImmutableList.of(State.TERMINATED), listener.getSt
ateHistory());
 }

public void testManualServiceFailWhileStarting() throws Exceptio
n {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync();
 service.notifyFailed(EXCEPTION);
 assertEquals(ImmutableList.of(State.STARTING, State.FAILED), li
stener.getStateHistory());
 }

public void testManualServiceFailWhileRunning() throws Exception
 {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync();
 service.notifyStarted();
 service.notifyFailed(EXCEPTION);
 assertEquals(ImmutableList.of(State.STARTING, State.RUNNING, St
ate.FAILED),
 listener.getStateHistory());
 }

public void testManualServiceFailWhileStopping() throws Exceptio
n {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync();
 service.notifyStarted();
 service.stopAsync();
 service.notifyFailed(EXCEPTION);
 assertEquals(ImmutableList.of(State.STARTING, State.RUNNING, St
ate.STOPPING, State.FAILED),
 listener.getStateHistory());
 }

public void testManualServiceUnrequestedStop() {
 ManualSwitchedService service = new ManualSwitchedService();

service.startAsync();

service.notifyStarted();
 assertEquals(State.RUNNING, service.state());
 assertTrue(service.isRunning());
 assertFalse(service.doStopCalled);

service.notifyStopped();
 assertEquals(State.TERMINATED, service.state());
 assertFalse(service.isRunning());
```

```
 assertFalse(service.doStopCalled);
 }

/**
 * The user of this service should call {@link #notifyStarted} a
nd {@link
 * #notifyStopped} after calling {@link #startAsync} and {@link
#stopAsync}.
 */
 private static class ManualSwitchedService extends AbstractServ
ice {
 boolean doStartCalled = false;
 boolean doStopCalled = false;

@Override protected void doStart() {
 assertFalse(doStartCalled);
 doStartCalled = true;
 }

@Override protected void doStop() {
 assertFalse(doStopCalled);
 doStopCalled = true;
 }
 }

public void testAwaitTerminated() throws Exception {
 final NoOpService service = new NoOpService();
 Thread waiter = new Thread() {
 @Override public void run() {
 service.awaitTerminated();
 }
 };
 waiter.start();
 service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());
 service.stopAsync();
 waiter.join(100); // ensure that the await in the other thread
is triggered
 assertFalse(waiter.isAlive());
 }

public void testAwaitTerminated_FailedService() throws Exception
 {
 final ManualSwitchedService service = new ManualSwitchedService
();
 final AtomicReference<Throwable> exception = Atomics.newReferen
ce();
 Thread waiter = new Thread() {
 @Override public void run() {
 try {
 service.awaitTerminated();
 fail("Expected an IllegalStateException");
 } catch (Throwable t) {
```

```
 exception.set(t);
 }
 }
 };
 waiter.start();
 service.startAsync();
 service.notifyStarted();
 assertEquals(State.RUNNING, service.state());
 service.notifyFailed(EXCEPTION);
 assertEquals(State.FAILED, service.state());
 waiter.join(100);
 assertFalse(waiter.isAlive());
 assertTrue(exception.get() instanceof IllegalStateException);
 assertEquals(EXCEPTION, exception.get().getCause());
 }

public void testThreadedServiceStartAndWaitStopAndWait() throws
Throwable {
 ThreadedService service = new ThreadedService();
 RecordingListener listener = RecordingListener.record(service);
 service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.awaitRunChecks();

service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());

throwIfSet(thrownByExecutionThread);
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.TERMINATED),
 listener.getStateHistory());
 }

public void testThreadedServiceStopIdempotence() throws Throwabl
e {
 ThreadedService service = new ThreadedService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.awaitRunChecks();

service.stopAsync();
 service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());

throwIfSet(thrownByExecutionThread);
 }
```

```
public void testThreadedServiceStopIdempotenceAfterWait()
 throws Throwable {
 ThreadedService service = new ThreadedService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.awaitRunChecks();

service.stopAsync().awaitTerminated();
 service.stopAsync();
 assertEquals(State.TERMINATED, service.state());

executionThread.join();

throwIfSet(thrownByExecutionThread);
 }

public void testThreadedServiceStopIdempotenceDoubleWait()
 throws Throwable {
 ThreadedService service = new ThreadedService();

service.startAsync().awaitRunning();
 assertEquals(State.RUNNING, service.state());

service.awaitRunChecks();

service.stopAsync().awaitTerminated();
 service.stopAsync().awaitTerminated();
 assertEquals(State.TERMINATED, service.state());

throwIfSet(thrownByExecutionThread);
 }

public void testManualServiceFailureIdempotence() {
 ManualSwitchedService service = new ManualSwitchedService();
 RecordingListener.record(service);
 service.startAsync();
 service.notifyFailed(new Exception("1"));
 service.notifyFailed(new Exception("2"));
 assertEquals("1", service.failureCause().getMessage());
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals("1", e.getCause().getMessage());
 }
 }

private class ThreadedService extends AbstractService {
 final CountDownLatch hasConfirmedIsRunning = new CountDownLatch
(1);
```

```
/*
 * The main test thread tries to stop() the service shortly afte
r
 * confirming that it is running. Meanwhile, the service itself
is trying
 * to confirm that it is running. If the main thread's stop() ca
ll happens
 * before it has the chance, the test will fail. To avoid this,
the main
 * thread calls this method, which waits until the service has p
erformed
 * its own "running" check.
 */
 void awaitRunChecks() throws InterruptedException {
 assertTrue("Service thread hasn't finished its checks. "
 + "Exception status (possibly stale): " + thrownByExecutionThre
ad,
 hasConfirmedIsRunning.await(10, SECONDS));
 }

@Override protected void doStart() {
 assertEquals(State.STARTING, state());
 invokeOnExecutionThreadForTest(new Runnable() {
 @Override public void run() {
 assertEquals(State.STARTING, state());
 notifyStarted();
 assertEquals(State.RUNNING, state());
 hasConfirmedIsRunning.countDown();
 }
 });
 }

@Override protected void doStop() {
 assertEquals(State.STOPPING, state());
 invokeOnExecutionThreadForTest(new Runnable() {
 @Override public void run() {
 assertEquals(State.STOPPING, state());
 notifyStopped();
 assertEquals(State.TERMINATED, state());
 }
 });
 }
 }

private void invokeOnExecutionThreadForTest(Runnable runnable) {
 executionThread = new Thread(runnable);
 executionThread.setUncaughtExceptionHandler(new UncaughtExcepti
onHandler() {
 @Override
 public void uncaughtException(Thread thread, Throwable e) {
 thrownByExecutionThread = e;
 }
```

```
 });
 executionThread.start();
 }

private static void throwIfSet(Throwable t) throws Throwable {
 if (t != null) {
 throw t;
 }
 }

public void testStopUnstartedService() throws Exception {
 NoOpService service = new NoOpService();
 RecordingListener listener = RecordingListener.record(service);

service.stopAsync();
 assertEquals(State.TERMINATED, service.state());

try {
 service.startAsync();
 fail();
 } catch (IllegalStateException expected) {}
 assertEquals(State.TERMINATED, Iterables.getOnlyElement(listene
r.getStateHistory()));
 }

public void testFailingServiceStartAndWait() throws Exception {
 StartFailingService service = new StartFailingService();
 RecordingListener listener = RecordingListener.record(service);

try {
 service.startAsync().awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(EXCEPTION, service.failureCause());
 assertEquals(EXCEPTION, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testFailingServiceStopAndWait_stopFailing() throws E
xception {
 StopFailingService service = new StopFailingService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync().awaitRunning();
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
```

```
 assertEquals(EXCEPTION, service.failureCause());
 assertEquals(EXCEPTION, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testFailingServiceStopAndWait_runFailing() throws Ex
ception {
 RunFailingService service = new RunFailingService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(EXCEPTION, service.failureCause());
 assertEquals(EXCEPTION, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testThrowingServiceStartAndWait() throws Exception {
 StartThrowingService service = new StartThrowingService();
 RecordingListener listener = RecordingListener.record(service);

try {
 service.startAsync().awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.exception, service.failureCause());
 assertEquals(service.exception, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testThrowingServiceStopAndWait_stopThrowing() throws
 Exception {
```

```
 StopThrowingService service = new StopThrowingService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync().awaitRunning();
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.exception, service.failureCause());
 assertEquals(service.exception, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.STOPPING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testThrowingServiceStopAndWait_runThrowing() throws
Exception {
 RunThrowingService service = new RunThrowingService();
 RecordingListener listener = RecordingListener.record(service);

service.startAsync();
 try {
 service.awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(service.exception, service.failureCause());
 assertEquals(service.exception, e.getCause());
 }
 assertEquals(
 ImmutableList.of(
 State.STARTING,
 State.RUNNING,
 State.FAILED),
 listener.getStateHistory());
 }

public void testFailureCause_throwsIfNotFailed() {
 StopFailingService service = new StopFailingService();
 try {
 service.failureCause();
 fail();
 } catch (IllegalStateException e) {
 // expected
 }
 service.startAsync().awaitRunning();
 try {
 service.failureCause();
 fail();
```

```
 } catch (IllegalStateException e) {
 // expected
 }
 try {
 service.stopAsync().awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(EXCEPTION, service.failureCause());
 assertEquals(EXCEPTION, e.getCause());
 }
 }

public void testAddListenerAfterFailureDoesntCauseDeadlock() thr
ows InterruptedException {
 final StartFailingService service = new StartFailingService();
 service.startAsync();
 assertEquals(State.FAILED, service.state());
 service.addListener(new RecordingListener(service), MoreExecuto
rs.sameThreadExecutor());
 Thread thread = new Thread() {
 @Override public void run() {
 // Internally stopAsync() grabs a lock, this could be any such
method on AbstractService.
 service.stopAsync();
 }
 };
 thread.start();
 thread.join(100);
 assertFalse(thread + " is deadlocked", thread.isAlive());
 }

public void testListenerDoesntDeadlockOnStartAndWaitFromRunning(
) throws Exception {
 final NoOpThreadedService service = new NoOpThreadedService();
 service.addListener(new Listener() {
 @Override public void running() {
 service.awaitRunning();
 }
 }, MoreExecutors.sameThreadExecutor());
 service.startAsync().awaitRunning(10, TimeUnit.MILLISECONDS);
 service.stopAsync();
 }

public void testListenerDoesntDeadlockOnStopAndWaitFromTerminate
d() throws Exception {
 final NoOpThreadedService service = new NoOpThreadedService();
 service.addListener(new Listener() {
 @Override public void terminated(State from) {
 service.stopAsync().awaitTerminated();
 }
 }, MoreExecutors.sameThreadExecutor());
 service.startAsync().awaitRunning();
```

```
Thread thread = new Thread() {
 @Override public void run() {
 service.stopAsync().awaitTerminated();
 }
 };
 thread.start();
 thread.join(100);
 assertFalse(thread + " is deadlocked", thread.isAlive());
 }

private static class NoOpThreadedService extends AbstractExecuti
onThreadService {
 final CountDownLatch latch = new CountDownLatch(1);
 @Override protected void run() throws Exception {
 latch.await();
 }
 @Override protected void triggerShutdown() {
 latch.countDown();
 }
 }

private static class StartFailingService extends AbstractService
 {
 @Override protected void doStart() {
 notifyFailed(EXCEPTION);
 }

@Override protected void doStop() {
 fail();
 }
 }

private static class RunFailingService extends AbstractService {
 @Override protected void doStart() {
 notifyStarted();
 notifyFailed(EXCEPTION);
 }

@Override protected void doStop() {
 fail();
 }
 }

private static class StopFailingService extends AbstractService
{
 @Override protected void doStart() {
 notifyStarted();
 }

@Override protected void doStop() {
 notifyFailed(EXCEPTION);
 }
 }
```

```
private static class StartThrowingService extends AbstractServic
e {

final RuntimeException exception = new RuntimeException("deliber
ate");

@Override protected void doStart() {
 throw exception;
 }

@Override protected void doStop() {
 fail();
 }
 }

private static class RunThrowingService extends AbstractService
{

final RuntimeException exception = new RuntimeException("deliber
ate");

@Override protected void doStart() {
 notifyStarted();
 throw exception;
 }

@Override protected void doStop() {
 fail();
 }
 }

private static class StopThrowingService extends AbstractService
 {

final RuntimeException exception = new RuntimeException("deliber
ate");

@Override protected void doStart() {
 notifyStarted();
 }

@Override protected void doStop() {
 throw exception;
 }
 }

private static class RecordingListener extends Listener {
 static RecordingListener record(Service service) {
 RecordingListener listener = new RecordingListener(service);
 service.addListener(listener, MoreExecutors.sameThreadExecutor(
));
 return listener;
```

```
 }

final Service service;

RecordingListener(Service service) {
 this.service = service;
 }

@GuardedBy("this")
 final List<State> stateHistory = Lists.newArrayList();
 final CountDownLatch completionLatch = new CountDownLatch(1);

ImmutableList<State> getStateHistory() throws Exception {
 completionLatch.await();
 synchronized (this) {
 return ImmutableList.copyOf(stateHistory);
 }
 }

@Override public synchronized void starting() {
 assertTrue(stateHistory.isEmpty());
 assertNotSame(State.NEW, service.state());
 stateHistory.add(State.STARTING);
 }

@Override public synchronized void running() {
 assertEquals(State.STARTING, Iterables.getOnlyElement(stateHist
ory));
 stateHistory.add(State.RUNNING);
 service.awaitRunning();
 assertNotSame(State.STARTING, service.state());
 }

@Override public synchronized void stopping(State from) {
 assertEquals(from, Iterables.getLast(stateHistory));
 stateHistory.add(State.STOPPING);
 if (from == State.STARTING) {
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException expected) {
 assertNull(expected.getCause());
 assertTrue(expected.getMessage().equals(
 "Expected the service to be RUNNING, but was STOPPING"));
 }
 }
 assertNotSame(from, service.state());
 }

@Override public synchronized void terminated(State from) {
 assertEquals(from, Iterables.getLast(stateHistory, State.NEW));
 stateHistory.add(State.TERMINATED);
 assertEquals(State.TERMINATED, service.state());
```

```
 if (from == State.NEW) {
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException expected) {
 assertNull(expected.getCause());
 assertTrue(expected.getMessage().equals(
 "Expected the service to be RUNNING, but was TERMINATED"));
 }
 }
 completionLatch.countDown();
 }

@Override public synchronized void failed(State from, Throwable
failure) {
 assertEquals(from, Iterables.getLast(stateHistory));
 stateHistory.add(State.FAILED);
 assertEquals(State.FAILED, service.state());
 assertEquals(failure, service.failureCause());
 if (from == State.STARTING) {
 try {
 service.awaitRunning();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(failure, e.getCause());
 }
 }
 try {
 service.awaitTerminated();
 fail();
 } catch (IllegalStateException e) {
 assertEquals(failure, e.getCause());
 }
 completionLatch.countDown();
 }
 }

public void testNotifyStartedWhenNotStarting() {
 AbstractService service = new DefaultService();
 try {
 service.notifyStarted();
 fail();
 } catch (IllegalStateException expected) {}
 }

public void testNotifyStoppedWhenNotRunning() {
 AbstractService service = new DefaultService();
 try {
 service.notifyStopped();
 fail();
 } catch (IllegalStateException expected) {}
 }
```

```
public void testNotifyFailedWhenNotStarted() {
 AbstractService service = new DefaultService();
 try {
 service.notifyFailed(new Exception());
 fail();
 } catch (IllegalStateException expected) {}
 }

public void testNotifyFailedWhenTerminated() {
 NoOpService service = new NoOpService();
 service.startAsync().awaitRunning();
 service.stopAsync().awaitTerminated();
 try {
 service.notifyFailed(new Exception());
 fail();
 } catch (IllegalStateException expected) {}
 }

private static class DefaultService extends AbstractService {
 @Override protected void doStart() {}
 @Override protected void doStop() {}
 }

private static final Exception EXCEPTION = new Exception();
}
<pre>
```

ServiceManagerTest

```
</pre>
/*
 * Copyright (C) 2012 The Guava Authors
 *
 * Licensed under the Apache License, Version 2.0 (the "License"
);
 * you may not use this file except in compliance with the Licen
se.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, so
ftware
 * distributed under the License is distributed on an "AS IS" BA
SIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or implied.
 * See the License for the specific language governing permissio
ns and
 * limitations under the License.
 */
```

```
package com.google.common.util.concurrent;

import static java.util.Arrays.asList;

import com.google.common.collect.ImmutableMap;
import com.google.common.collect.ImmutableSet;
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
import com.google.common.testing.NullPointerTester;
import com.google.common.testing.TestLogHandler;
import com.google.common.util.concurrent.ServiceManager.Listener
;

import junit.framework.TestCase;

import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.Set;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Executor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.util.logging.Formatter;
import java.util.logging.Level;
import java.util.logging.LogRecord;
import java.util.logging.Logger;

/**
 * Tests for {@link ServiceManager}.
 *
 * @author Luke Sandberg
 * @author Chris Nokleberg
 */
public class ServiceManagerTest extends TestCase {

private static class NoOpService extends AbstractService {
 @Override protected void doStart() {
 notifyStarted();
 }

@Override protected void doStop() {
 notifyStopped();
 }
 }

/*
 * A NoOp service that will delay the startup and shutdown notif
ication for a configurable amount
 * of time.
 */
 private static class NoOpDelayedSerivce extends NoOpService {
 private long delay;
```

```
public NoOpDelayedSerivce(long delay) {
 this.delay = delay;
 }

@Override protected void doStart() {
 new Thread() {
 @Override public void run() {
 Uninterruptibles.sleepUninterruptibly(delay, TimeUnit.MILLISECO
NDS);
 notifyStarted();
 }
 }.start();
 }

@Override protected void doStop() {
 new Thread() {
 @Override public void run() {
 Uninterruptibles.sleepUninterruptibly(delay, TimeUnit.MILLISECO
NDS);
 notifyStopped();
 }
 }.start();
 }
 }

private static class FailStartService extends NoOpService {
 @Override protected void doStart() {
 notifyFailed(new IllegalStateException("failed"));
 }
 }

private static class FailRunService extends NoOpService {
 @Override protected void doStart() {
 super.doStart();
 notifyFailed(new IllegalStateException("failed"));
 }
 }

private static class FailStopService extends NoOpService {
 @Override protected void doStop() {
 notifyFailed(new IllegalStateException("failed"));
 }
 }

public void testServiceStartupTimes() {
 Service a = new NoOpDelayedSerivce(150);
 Service b = new NoOpDelayedSerivce(353);
 ServiceManager serviceManager = new ServiceManager(asList(a, b)
);
 serviceManager.startAsync().awaitHealthy();
 ImmutableMap<Service, Long> startupTimes = serviceManager.start
upTimes();
```

```
 assertEquals(2, startupTimes.size());
 assertTrue(startupTimes.get(a) >= 150);
 assertTrue(startupTimes.get(b) >= 353);
 }

public void testServiceStartStop() {
 Service a = new NoOpService();
 Service b = new NoOpService();
 ServiceManager manager = new ServiceManager(asList(a, b));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
 assertState(manager, Service.State.NEW, a, b);
 assertFalse(manager.isHealthy());
 manager.startAsync().awaitHealthy();
 assertState(manager, Service.State.RUNNING, a, b);
 assertTrue(manager.isHealthy());
 assertTrue(listener.healthyCalled);
 assertFalse(listener.stoppedCalled);
 assertTrue(listener.failedServices.isEmpty());
 manager.stopAsync().awaitStopped();
 assertState(manager, Service.State.TERMINATED, a, b);
 assertFalse(manager.isHealthy());
 assertTrue(listener.stoppedCalled);
 assertTrue(listener.failedServices.isEmpty());
 }

public void testFailStart() throws Exception {
 Service a = new NoOpService();
 Service b = new FailStartService();
 Service c = new NoOpService();
 Service d = new FailStartService();
 Service e = new NoOpService();
 ServiceManager manager = new ServiceManager(asList(a, b, c, d,
e));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
 assertState(manager, Service.State.NEW, a, b, c, d, e);
 try {
 manager.startAsync().awaitHealthy();
 fail();
 } catch (IllegalStateException expected) {
 }
 assertFalse(listener.healthyCalled);
 assertState(manager, Service.State.RUNNING, a, c, e);
 assertEquals(ImmutableSet.of(b, d), listener.failedServices);
 assertState(manager, Service.State.FAILED, b, d);
 assertFalse(manager.isHealthy());

manager.stopAsync().awaitStopped();
 assertFalse(manager.isHealthy());
 assertFalse(listener.healthyCalled);
 assertTrue(listener.stoppedCalled);
 }
```

```
public void testFailRun() throws Exception {
 Service a = new NoOpService();
 Service b = new FailRunService();
 ServiceManager manager = new ServiceManager(asList(a, b));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
 assertState(manager, Service.State.NEW, a, b);
 try {
 manager.startAsync().awaitHealthy();
 fail();
 } catch (IllegalStateException expected) {
 }
 assertTrue(listener.healthyCalled);
 assertEquals(ImmutableSet.of(b), listener.failedServices);

manager.stopAsync().awaitStopped();
 assertState(manager, Service.State.FAILED, b);
 assertState(manager, Service.State.TERMINATED, a);

assertTrue(listener.stoppedCalled);
 }

public void testFailStop() throws Exception {
 Service a = new NoOpService();
 Service b = new FailStopService();
 Service c = new NoOpService();
 ServiceManager manager = new ServiceManager(asList(a, b, c));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);

manager.startAsync().awaitHealthy();
 assertTrue(listener.healthyCalled);
 assertFalse(listener.stoppedCalled);
 manager.stopAsync().awaitStopped();

assertTrue(listener.stoppedCalled);
 assertEquals(ImmutableSet.of(b), listener.failedServices);
 assertState(manager, Service.State.FAILED, b);
 assertState(manager, Service.State.TERMINATED, a, c);
 }

public void testToString() throws Exception {
 Service a = new NoOpService();
 Service b = new FailStartService();
 ServiceManager manager = new ServiceManager(asList(a, b));
 String toString = manager.toString();
 assertTrue(toString.contains("NoOpService"));
 assertTrue(toString.contains("FailStartService"));
 }

public void testTimeouts() throws Exception {
 Service a = new NoOpDelayedSerivce(50);
```

```
 ServiceManager manager = new ServiceManager(asList(a));
 manager.startAsync();
 try {
 manager.awaitHealthy(1, TimeUnit.MILLISECONDS);
 fail();
 } catch (TimeoutException expected) {
 }
 manager.awaitHealthy(100, TimeUnit.MILLISECONDS); // no excepti
on thrown

manager.stopAsync();
 try {
 manager.awaitStopped(1, TimeUnit.MILLISECONDS);
 fail();
 } catch (TimeoutException expected) {
 }
 manager.awaitStopped(100, TimeUnit.MILLISECONDS); // no excepti
on thrown
 }

/**
 * This covers a case where if the last service to stop failed t
hen the stopped callback would
 * never be called.
 */
 public void testSingleFailedServiceCallsStopped() {
 Service a = new FailStartService();
 ServiceManager manager = new ServiceManager(asList(a));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
 try {
 manager.startAsync().awaitHealthy();
 fail();
 } catch (IllegalStateException expected) {
 }
 assertTrue(listener.stoppedCalled);
 }

/**
 * This covers a bug where listener.healthy would get called whe
n a single service failed during
 * startup (it occurred in more complicated cases also).
 */
 public void testFailStart_singleServiceCallsHealthy() {
 Service a = new FailStartService();
 ServiceManager manager = new ServiceManager(asList(a));
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener);
 try {
 manager.startAsync().awaitHealthy();
 fail();
 } catch (IllegalStateException expected) {
 }
```

```
 assertFalse(listener.healthyCalled);
 }

/**
 * This covers a bug where if a listener was installed that woul
d stop the manager if any service
 * fails and something failed during startup before service.star
t was called on all the services,
 * then awaitStopped would deadlock due to an IllegalStateExcept
ion that was thrown when trying to
 * stop the timer(!).
 */
 public void testFailStart_stopOthers() throws TimeoutException
{
 Service a = new FailStartService();
 Service b = new NoOpService();
 final ServiceManager manager = new ServiceManager(asList(a, b))
;
 manager.addListener(new Listener() {
 @Override public void failure(Service service) {
 manager.stopAsync();
 }});
 manager.startAsync();
 manager.awaitStopped(10, TimeUnit.MILLISECONDS);
 }

private static void assertState(
 ServiceManager manager, Service.State state, Service... service
s) {
 Collection<Service> managerServices = manager.servicesByState()
.get(state);
 for (Service service : services) {
 assertEquals(service.toString(), state, service.state());
 assertEquals(service.toString(), service.isRunning(), state ==
Service.State.RUNNING);
 assertTrue(managerServices + " should contain " + service, mana
gerServices.contains(service));
 }
 }

/**
 * This is for covering a case where the ServiceManager would be
have strangely if constructed
 * with no service under management. Listeners would never fire
because the ServiceManager was
 * healthy and stopped at the same time. This test ensures that
listeners fire and isHealthy
 * makes sense.
 */
 public void testEmptyServiceManager() {
 Logger logger = Logger.getLogger(ServiceManager.class.getName()
);
 logger.setLevel(Level.FINEST);
```

```
 TestLogHandler logHandler = new TestLogHandler();
 logger.addHandler(logHandler);
 ServiceManager manager = new ServiceManager(Arrays.<Service>asL
ist());
 RecordingListener listener = new RecordingListener();
 manager.addListener(listener, MoreExecutors.sameThreadExecutor(
));
 manager.startAsync().awaitHealthy();
 assertTrue(manager.isHealthy());
 assertTrue(listener.healthyCalled);
 assertFalse(listener.stoppedCalled);
 assertTrue(listener.failedServices.isEmpty());
 manager.stopAsync().awaitStopped();
 assertFalse(manager.isHealthy());
 assertTrue(listener.stoppedCalled);
 assertTrue(listener.failedServices.isEmpty());
 // check that our NoOpService is not directly observable via an
y of the inspection methods or
 // via logging.
 assertEquals("ServiceManager{services=[]}", manager.toString())
;
 assertTrue(manager.servicesByState().isEmpty());
 assertTrue(manager.startupTimes().isEmpty());
 Formatter logFormatter = new Formatter() {
 @Override public String format(LogRecord record) {
 return formatMessage(record);
 }
 };
 for (LogRecord record : logHandler.getStoredLogRecords()) {
 assertFalse(logFormatter.format(record).contains("NoOpService")
);
 }
 }

/**
 * This is for a case where a long running Listener using the sa
meThreadListener could deadlock
 * another thread calling stopAsync().
 */

public void testListenerDeadlock() throws InterruptedException {
 final CountDownLatch failEnter = new CountDownLatch(1);
 Service failRunService = new AbstractService() {
 @Override protected void doStart() {
 new Thread() {
 @Override public void run() {
 notifyStarted();
 notifyFailed(new Exception("boom"));
 }
 }.start();
 }
 @Override protected void doStop() {
 notifyStopped();
```

```
 }
 };
 final ServiceManager manager = new ServiceManager(
 Arrays.asList(failRunService, new NoOpService()));
 manager.addListener(new ServiceManager.Listener() {
 @Override public void failure(Service service) {
 failEnter.countDown();
 // block forever!
 Uninterruptibles.awaitUninterruptibly(new CountDownLatch(1));
 }
 }, MoreExecutors.sameThreadExecutor());
 // We do not call awaitHealthy because, due to races, that meth
od may throw an exception. But
 // we really just want to wait for the thread to be in the fail
ure callback so we wait for that
 // explicitly instead.
 manager.startAsync();
 failEnter.await();
 assertFalse("State should be updated before calling listeners",
 manager.isHealthy());
 // now we want to stop the services.
 Thread stoppingThread = new Thread() {
 @Override public void run() {
 manager.stopAsync().awaitStopped();
 }
 };
 stoppingThread.start();
 // this should be super fast since the only non stopped service
 is a NoOpService
 stoppingThread.join(1000);
 assertFalse("stopAsync has deadlocked!.", stoppingThread.isAliv
e());
 }

/**
 * Catches a bug where when constructing a service manager faile
d, later interactions with the
 * service could cause IllegalStateExceptions inside the partial
ly constructed ServiceManager.
 * This ISE wouldn't actually bubble up but would get logged by
ExecutionQueue. This obfuscated
 * the original error (which was not constructing ServiceManager
 correctly).
 */
 public void testPartiallyConstructedManager() {
 Logger logger = Logger.getLogger("global");
 logger.setLevel(Level.FINEST);
 TestLogHandler logHandler = new TestLogHandler();
 logger.addHandler(logHandler);
 NoOpService service = new NoOpService();
 service.startAsync();
 try {
 new ServiceManager(Arrays.asList(service));
```

```
 fail();
 } catch (IllegalArgumentException expected) {}
 service.stopAsync();
 // Nothing was logged!
 assertEquals(0, logHandler.getStoredLogRecords().size());
 }

public void testPartiallyConstructedManager_transitionAfterAddLi
stenerBeforeStateIsReady() {
 // The implementation of this test is pretty sensitive to the i
mplementation ![:(](http://ifeve.com/wp-includes/images/smilies/
frownie.png) but we want to
 // ensure that if weird things happen during construction then
we get exceptions.
 final NoOpService service1 = new NoOpService();
 // This service will start service1 when addListener is called.
 This simulates service1 being
 // started asynchronously.
 Service service2 = new Service() {
 final NoOpService delegate = new NoOpService();
 @Override public final void addListener(Listener listener, Exec
utor executor) {
 service1.startAsync();
 delegate.addListener(listener, executor);
 }
 // Delegates from here on down
 @Override public final Service startAsync() {
 return delegate.startAsync();
 }

@Override public final Service stopAsync() {
 return delegate.stopAsync();
 }

@Override public final ListenableFuture<State> start() {
 return delegate.start();
 }

@Override public final ListenableFuture<State> stop() {
 return delegate.stop();
 }

@Override public State startAndWait() {
 return delegate.startAndWait();
 }

@Override public State stopAndWait() {
 return delegate.stopAndWait();
 }

@Override public final void awaitRunning() {
 delegate.awaitRunning();
 }
```

```
@Override public final void awaitRunning(long timeout, TimeUnit unit)
 throws TimeoutException {
 delegate.awaitRunning(timeout, unit);
 }

@Override public final void awaitTerminated() {
 delegate.awaitTerminated();
 }

@Override public final void awaitTerminated(long timeout, TimeUnit unit)
 throws TimeoutException {
 delegate.awaitTerminated(timeout, unit);
 }

@Override public final boolean isRunning() {
 return delegate.isRunning();
 }

@Override public final State state() {
 return delegate.state();
 }

@Override public final Throwable failureCause() {
 return delegate.failureCause();
 }
 };
 try {
 new ServiceManager(Arrays.asList(service1, service2));
 fail();
 } catch (IllegalArgumentException expected) {
 assertTrue(expected.getMessage().contains("started transitioning asynchronously"));
 }
 }

/**
 * This test is for a case where two Service.Listener callbacks for the same service would call
 * transitionService in the wrong order due to a race. Due to the fact that it is a race this
 * test isn't guaranteed to expose the issue, but it is at least likely to become flaky if the
 * race sneaks back in, and in this case flaky means something is definitely wrong.
 *
 * <p>Before the bug was fixed this test would fail at least 30% of the time.
 */

public void testTransitionRace() throws TimeoutException {
```

```
 for (int k = 0; k < 1000; k++) {
 List<Service> services = Lists.newArrayList();
 for (int i = 0; i < 5; i++) {
 services.add(new SnappyShutdownService(i));
 }
 ServiceManager manager = new ServiceManager(services);
 manager.startAsync().awaitHealthy();
 manager.stopAsync().awaitStopped(1, TimeUnit.SECONDS);
 }
 }

/**
 * This service will shutdown very quickly after stopAsync is ca
lled and uses a background thread
 * so that we know that the stopping() listeners will execute on
 a different thread than the
 * terminated() listeners.
 */
 private static class SnappyShutdownService extends AbstractExec
utionThreadService {
 final int index;
 final CountDownLatch latch = new CountDownLatch(1);

SnappyShutdownService(int index) {
 this.index = index;
 }

@Override protected void run() throws Exception {
 latch.await();
 }

@Override protected void triggerShutdown() {
 latch.countDown();
 }

@Override protected String serviceName() {
 return this.getClass().getSimpleName() + "[" + index + "]";
 }
 }

public void testNulls() {
 ServiceManager manager = new ServiceManager(Arrays.<Service>asL
ist());
 new NullPointerTester()
 .setDefault(ServiceManager.Listener.class, new RecordingListene
r())
 .testAllPublicInstanceMethods(manager);
 }

private static final class RecordingListener extends ServiceMana
ger.Listener {
 volatile boolean healthyCalled;
 volatile boolean stoppedCalled;
```

```
 final Set<Service> failedServices = Sets.newConcurrentHashSet()
;

@Override public void healthy() {
 healthyCalled = true;
 }

@Override public void stopped() {
 stoppedCalled = true;
 }

@Override public void failure(Service service) {
 failedServices.add(service);
 }
 }
}
<pre>
```

6-字符串处理：分割，连接，填充

原文链接 译文链接 译者：沈义扬，校对：丁一

连接器[Joiner]

用分隔符把字符串序列连接起来也可能会遇上不必要的麻烦。如果字符串序列中含有null，那连接操作会更难。Fluent风格的`Joiner`让连接字符串更简单。

```
Joiner joiner = Joiner.on("; ").skipNulls();
return joiner.join("Harry", null, "Ron", "Hermione");
```

上述代码返回”Harry; Ron; Hermione”。另外，useForNull(String)方法可以给定某个字符串来替换null，而不像skipNulls()方法是直接忽略null。 Joiner也可以用来连接对象类型，在这种情况下，它会把对象的toString()值连接起来。

```
Joiner.on(",").join(Arrays.asList(1, 5, 7)); // returns "1,5,7"
```

警告：*joiner*实例总是不可变的。用来定义*joiner*目标语义的配置方法总会返回一个新的*joiner*实例。这使得*joiner*实例都是线程安全的，你可以将其定义为*static final*常量。

拆分器[Splitter]

JDK内建的字符串拆分工具有一些古怪的特性。比如，String.split悄悄丢弃了尾部的分隔符。 问题：”,a,,b,”.split(“,”)返回？

1. “”, “a”, “”, “b”, “”
2. null, “a”, null, “b”, null
3. “a”, null, “b”
4. “a”, “b”
5. 以上都不对

正确答案是5：””, “a”, “”, “b”。只有尾部的空字符串被忽略了。 `Splitter`使用令人放心的、直白的流畅API模式对这些混乱的特性作了完全的掌控。

```
Splitter.on(',')
        .trimResults()
        .omitEmptyStrings()
        .split("foo,bar,,   qux");
```

上述代码返回Iterable<String>，其中包含"foo"、"bar"和"qux"。Splitter可以被设置为按照任何模式、字符、字符串或字符匹配器拆分。

拆分器工厂

方法	描述	范例
`Splitter.on(char)`	按单个字符拆分	Splitter.on(';')
`Splitter.on(CharMatcher)`	按字符匹配器拆分	Splitter.on(CharMatcher.BREAKING
`Splitter.on(Pattern)` `Splitter.onPattern(String)`	按正则表达式拆分	Splitter.onPattern("\r?\n")
`Splitter.on(String)`	按字符串拆分	Splitter.on(", ")
`Splitter.fixedLength(int)`	按固定长度拆分；最后一段可能比给定长度短，但不会为空。	Splitter.fixedLength(3)

拆分器修饰符

方法	描述
`omitEmptyStrings()`	从结果中自动忽略空字符串
`trimResults()`	移除结果字符串的前导空白和尾部空白
`trimResults(CharMatcher)`	给定匹配器，移除结果字符串的前导匹配字符和尾部匹配字符
`limit(int)`	限制拆分出的字符串数量

如果你想要拆分器返回List，只要使用Lists.newArrayList(splitter.split(string))或类似方法。 警告：*splitter*实例总是不可变的。用来定义*splitter*目标语义的配置方法总会返回一个新的*splitter*实例。这使得*splitter*实例都是线程安全的，你可以将其定义为*static final*常量。

字符匹配器[CharMatcher]

在以前的Guava版本中，StringUtil类疯狂地膨胀，其拥有很多处理字符串的方法：allAscii、collapse、collapseControlChars、collapseWhitespace、indexOfChars、lastIndexNotOf、numSharedChars、removeChars、removeCrLf、replaceChars、retainAllChars、strip、stripAndCollapse、stripNonDigits。 所有这些方法指向两个概念上的问题：

1. 怎么才算匹配字符？
2. 如何处理这些匹配字符？

为了收拾这个泥潭，我们开发了CharMatcher。

直观上，你可以认为一个CharMatcher实例代表着某一类字符，如数字或空白字符。事实上来说，CharMatcher实例就是对字符的布尔判断——CharMatcher确实也实现了Predicate<Character>——但类似"所有空白字符"或"所有小写字母"的需求太普遍了，Guava因此创建了这一API。

然而使用CharMatcher的好处更在于它提供了一系列方法，让你对字符作特定类型的操作：修剪[trim]、折叠[collapse]、移除[remove]、保留[retain]等等。CharMatcher实例首先代表概念1：怎么才算匹配字符？然后它还提供了很多操作概念2：如何处理这些匹配字符？这样的设计使得API复杂度的线性增加可以带来灵活性和功能两方面的增长。

```
String noControl = CharMatcher.JAVA_ISO_CONTROL.removeFrom(strin
g); //移除control字符
String theDigits = CharMatcher.DIGIT.retainFrom(string); //只保留
数字字符
String spaced = CharMatcher.WHITESPACE.trimAndCollapseFrom(strin
g, ' ');
//去除两端的空格，并把中间的连续空格替换成单个空格
String noDigits = CharMatcher.JAVA_DIGIT.replaceFrom(string, "*"
); //用*号替换所有数字
String lowerAndDigit = CharMatcher.JAVA_DIGIT.or(CharMatcher.JAV
A_LOWER_CASE).retainFrom(string);
// 只保留数字和小写字母
```

注：CharMatcher只处理char类型代表的字符；它不能理解0x10000到0x10FFFF的Unicode 增补字符。这些逻辑字符以代理对[surrogate pairs]的形式编码进字符串，而CharMatcher只能将这种逻辑字符看待成两个独立的字符。

获取字符匹配器

CharMatcher中的常量可以满足大多数字符匹配需求：

`ANY`	`NONE`	`WHITESPACE`	[illegible]
`INVISIBLE`	`DIGIT`	`JAVA_LETTER`	[illegible]
`JAVA_LETTER_OR_DIGIT`	`JAVA_ISO_CONTROL`	`JAVA_LOWER_CASE`	[illegible]
`ASCII`	`SINGLE_WIDTH`		

其他获取字符匹配器的常见方法包括：

方法	描述
`anyOf(CharSequence)`	枚举匹配字符。如CharMatcher.anyOf(“aeiou”)匹配小写英语元音
`is(char)`	给定单一字符匹配。
`inRange(char, char)`	给定字符范围匹配，如CharMatcher.inRange(‘a’, ‘z’)

此外，CharMatcher还有 `negate()` 、 `and(CharMatcher)` 和 `or(CharMatcher)` 方法。

使用字符匹配器

CharMatcher提供了多种多样的方法操作CharSequence中的特定字符。其中最常用的罗列如下：

方法	描述
`collapseFrom(CharSequence, char)`	把每组连续的匹配字符替换 字符。如 WHITESPACE.collapseFror ‘ ’)把字符串中的连续空白字 为单个空格。
`matchesAllOf(CharSequence)`	测试是否字符序列中的所有 匹配。
`removeFrom(CharSequence)`	从字符序列中移除所有匹配
`retainFrom(CharSequence)`	在字符序列中保留匹配字符 其他字符。
`trimFrom(CharSequence)`	移除字符序列的前导匹配字 部匹配字符。
`replaceFrom(CharSequence, CharSequence)`	用特定字符序列替代匹配字

所有这些方法返回String，除了matchesAllOf返回的是boolean。

字符集[Charsets]

不要这样做字符集处理：

```
try {
    bytes = string.getBytes("UTF-8");
} catch (UnsupportedEncodingException e) {
    // how can this possibly happen?
    throw new AssertionError(e);
}
```

试试这样写：

```
bytes = string.getBytes(Charsets.UTF_8);
```

`Charsets` 针对所有Java平台都要保证支持的六种字符集提供了常量引用。尝试使用这些常量，而不是通过名称获取字符集实例。

大小写格式[CaseFormat]

CaseFormat被用来方便地在各种ASCII大小写规范间转换字符串——比如，编程语言的命名规范。CaseFormat支持的格式如下：

格式	范例
`LOWER_CAMEL`	lowerCamel
`LOWER_HYPHEN`	lower-hyphen
`LOWER_UNDERSCORE`	lower_underscore
`UPPER_CAMEL`	UpperCamel
`UPPER_UNDERSCORE`	UPPER_UNDERSCORE

CaseFormat的用法很直接：

```
CaseFormat.UPPER_UNDERSCORE.to(CaseFormat.LOWER_CAMEL, "CONSTANT
_NAME")); // returns "constantName"
```

我们CaseFormat在某些时候尤其有用，比如编写代码生成器的时候。

7-原生类型

原文链接 译文链接 译者：沈义扬，校对：丁一

概述

Java的原生类型就是指基本类型：byte、short、int、long、float、double、char和boolean。

在从*Guava*查找原生类型方法之前，可以先查查***Arrays***类，或者对应的基础类型包装类，如***Integer***。

原生类型不能当作对象或泛型的类型参数使用，这意味着许多通用方法都不能应用于它们。Guava提供了若干通用工具，包括原生类型数组与集合API的交互，原生类型和字节数组的相互转换，以及对某些原生类型的无符号形式的支持。

原生类型	Guava**工具类（都在**com.google.common.primitives**包**）
byte	`Bytes`, `SignedBytes`, `UnsignedBytes`
short	`Shorts`
int	`Ints`, `UnsignedInteger`, `UnsignedInts`
long	`Longs`, `UnsignedLong`, `UnsignedLongs`
float	`Floats`
double	`Doubles`
char	`Chars`
boolean	`Booleans`

Bytes工具类没有定义任何区分有符号和无符号字节的方法，而是把它们都放到了SignedBytes和UnsignedBytes工具类中，因为字节类型的符号性比起其它类型要略微含糊一些。

int和long的无符号形式方法在UnsignedInts和UnsignedLongs类中，但由于这两个类型的大多数用法都是有符号的，Ints和Longs类按照有符号形式处理方法的输入参数。

此外，Guava为int和long的无符号形式提供了包装类，即UnsignedInteger和UnsignedLong，以帮助你使用类型系统，以极小的性能消耗对有符号和无符号值进行强制转换。

在本章下面描述的方法签名中，我们用Wrapper表示JDK包装类，prim表示原生类型。（Prims表示相应的Guava工具类。）

原生类型数组工具

原生类型数组是处理原生类型集合的最有效方式（从内存和性能双方面考虑）。Guava为此提供了许多工具方法。

方法签名	描述	类似方法
List<Wrapper> asList(prim… backingArray)	把数组转为相应包装类的List	Arrays.asList
prim[] toArray(Collection<Wrapper> collection)	把集合拷贝为数组，和collection.toArray()一样线程安全	Collection.toArray()
prim[] concat(prim[]… arrays)	串联多个原生类型数组	Iterables.concat
boolean contains(prim[] array, prim target)	判断原生类型数组是否包含给定值	Collection.contains
int indexOf(prim[] array, prim target)	给定值在数组中首次出现处的索引，若不包含此值返回-1	List.indexOf
int lastIndexOf(prim[] array, prim target)	给定值在数组最后出现的索引，若不包含此值返回-1	List.lastIndexOf
prim min(prim… array)	数组中最小的值	Collections.min
prim max(prim… array)	数组中最大的值	Collections.max

String join(String separator, prim… array)	把数组用给定分隔符连接为字符串	Joiner.on(separator).join
Comparator<prim[]> lexicographicalComparator()	按字典序比较原生类型数组的Comparator	Ordering.natural().lexicog

*符号无关方法存在于Bytes, Shorts, Ints, Longs, Floats, Doubles, Chars, Booleans。而UnsignedInts, UnsignedLongs, SignedBytes, 或UnsignedBytes不存在。

*符号相关方法存在于SignedBytes, UnsignedBytes, Shorts, Ints, Longs, Floats, Doubles, Chars, Booleans, UnsignedInts, UnsignedLongs。而Bytes不存在。

通用工具方法

Guava为原生类型提供了若干JDK6没有的工具方法。但请注意，其中某些方法已经存在于JDK7中。

方法签名	描述	可用性
int compare(prim a, prim b)	传统的Comparator.compare方法，但针对原生类型。JDK7的原生类型包装类也提供这样的方法	符号相关
prim checkedCast(long value)	把给定long值转为某一原生类型，若给定值不符合该原生类型，则抛出IllegalArgumentException	仅适用于符号相关的整型*
prim saturatedCast(long value)	把给定long值转为某一原生类型，若给定值不符合则使用最接近的原生类型值	仅适用于符号相关的整型

*这里的整型包括byte, short, int, long。不包括char, boolean, float, 或double。

**译者注：不符合主要是指*long*值超出*prim*类型的范围，比如过大的*long*超出*int*范围。

注：com.google.common.math.DoubleMath提供了舍入double的方法，支持多种舍入模式。相见第12章的"浮点数运算"。

字节转换方法

Guava提供了若干方法，用来把原生类型按大字节序与字节数组相互转换。所有这些方法都是符号无关的，此外Booleans没有提供任何下面的方法。

方法或字段签名	描述
int BYTES	常量：表示该原生类型需要的字节数
prim fromByteArray(byte[] bytes)	使用字节数组的前Prims.BYTES个字节，按大字节序返回原生类型值；如果bytes.length <= Prims.BYTES，抛出IAE
prim fromBytes(byte b1, …, byte bk)	接受Prims.BYTES个字节参数，按大字节序返回原生类型值
byte[] toByteArray(prim value)	按大字节序返回value的字节数组

无符号支持

JDK原生类型包装类提供了针对有符号类型的方法，而UnsignedInts和UnsignedLongs工具类提供了相应的无符号通用方法。UnsignedInts和UnsignedLongs直接处理原生类型：使用时，由你自己保证只传入了无符号类型的值。

此外，对int和long，Guava提供了无符号包装类（UnsignedInteger和UnsignedLong），来帮助你以极小的性能消耗，对有符号和无符号类型进行强制转换。

无符号通用工具方法

JDK的原生类型包装类提供了有符号形式的类似方法。

方法签名
`int UnsignedInts.parseUnsignedInt(String)` `long UnsignedLongs.pars`

`int UnsignedInts.parseUnsignedInt(String string, int radix)`	`long`
`String UnsignedInts.toString(int)`	`String UnsignedLongs.toString(l`
`String UnsignedInts.toString(int value, int radix)`	`String Unsigne`

无符号包装类

无符号包装类包含了若干方法，让使用和转换更容易。

方法签名	说明
UnsignedPrim add(UnsignedPrim), subtract, multiply, divide, remainder	简单算术运算
UnsignedPrim valueOf(BigInteger)	按给定BigInteger返回无符号对象，若BigInteger为负或不匹配，抛出IAE
UnsignedPrim valueOf(long)	按给定long返回无符号对象，若long为负或不匹配，抛出IAE
UnsignedPrim asUnsigned(prim value)	把给定的值当作无符号类型。例如，UnsignedInteger.asUnsigned(1<<31)的值为2³¹,尽管1<<31当作int时是负的
BigInteger bigIntegerValue()	用BigInteger返回该无符号对象的值
toString(), toString(int radix)	返回无符号值的字符串表示

8-区间

原文链接 译文链接 译文：沈义扬

范例

```
List scores;
Iterable belowMedian =Iterables.filter(scores,Range.lessThan(med
ian));
...
Range validGrades = Range.closed(1, 12);
for(int grade : ContiguousSet.create(validGrades, DiscreteDomain
.integers())) {
    ...
}
```

简介

区间，有时也称为范围，是特定域中的凸性（非正式说法为连续的或不中断的）部分。在形式上，凸性表示对a<=b<=c, range.contains(a)且range.contains(c)意味着range.contains(b)。

区间可以延伸至无限——例如，范围"x>3"包括任意大于3的值——也可以被限制为有限，如" 2<=x<5"。Guava用更紧凑的方法表示范围，有数学背景的程序员对此是耳熟能详的：

- (a..b) = {x | a < x < b}
- [a..b] = {x | a <= x <= b}
- [a..b) = {x | a <= x < b}
- (a..b] = {x | a < x <= b}
- (a..+∞) = {x | x > a}
- [a..+∞) = {x | x >= a}
- (-∞..b) = {x | x < b}
- (-∞..b] = {x | x <= b}
- (-∞..+∞) = 所有值

上面的a、b称为端点 。为了提高一致性，Guava中的Range要求上端点不能小于下端点。上下端点有可能是相等的，但要求区间是闭区间或半开半闭区间（至少有一个端点是包含在区间中的）：

- [a..a]：单元素区间
- [a..a); (a..a]：空区间，但它们是有效的
- (a..a)：无效区间

Guava用类型Range<C>表示区间。所有区间实现都是不可变类型。

构建区间

区间实例可以由Range类的静态方法获取：

(a..b)	open(C, C)
[a..b]	closed(C, C)
[a..b)	closedOpen(C, C)
(a..b]	openClosed(C, C)
(a..+∞)	greaterThan(C)
[a..+∞)	atLeast(C)
(-∞..b)	lessThan(C)
(-∞..b]	atMost(C)
(-∞..+∞)	all()

```
Range.closed("left", "right"); //字典序在"left"和"right"之间的字符串
，闭区间
Range.lessThan(4.0); //严格小于4.0的double值
```

此外，也可以明确地指定边界类型来构造区间：

有界区间	range(C, BoundType, C, BoundType))
无上界区间：((a..+∞) 或[a..+∞))	downTo(C, BoundType)
无下界区间：((-∞..b) 或(-∞..b])	upTo(C, BoundType)

这里的BoundType是一个枚举类型，包含CLOSED和OPEN两个值。

```
Range.downTo(4, boundType);// (a..+∞)或[a..+∞)，取决于boundType
Range.range(1, CLOSED, 4, OPEN);// [1..4)，等同于Range.closedOpen
(1, 4)
```

区间运算

Range的基本运算是它的contains(C) 方法，和你期望的一样，它用来区间判断是否包含某个值。此外，Range实例也可以当作Predicate，并且在函数式编程中使用（译者注：见第4章）。任何Range实例也都支持containsAll(Iterable<? extends

C>)方法：

```
Range.closed(1, 3).contains(2);//return true
Range.closed(1, 3).contains(4);//return false
Range.lessThan(5).contains(5); //return false
Range.closed(1, 4).containsAll(Ints.asList(1, 2, 3)); //return t
rue
```

查询运算

Range类提供了以下方法来 查看区间的端点：

- hasLowerBound()和hasUpperBound()：判断区间是否有特定边界，或是无限的；
- lowerBoundType()和upperBoundType()：返回区间边界类型，CLOSED或OPEN；如果区间没有对应的边界，抛出IllegalStateException；
- lowerEndpoint()和upperEndpoint()：返回区间的端点值；如果区间没有对应的边界，抛出IllegalStateException；
- isEmpty()：判断是否为空区间。

```
Range.closedOpen(4, 4).isEmpty(); // returns true
Range.openClosed(4, 4).isEmpty(); // returns true
Range.closed(4, 4).isEmpty(); // returns false
Range.open(4, 4).isEmpty(); // Range.open throws IllegalArgument
Exception
Range.closed(3, 10).lowerEndpoint(); // returns 3
Range.open(3, 10).lowerEndpoint(); // returns 3
Range.closed(3, 10).lowerBoundType(); // returns CLOSED
Range.open(3, 10).upperBoundType(); // returns OPEN
```

关系运算

包含[enclose]

区间之间的最基本关系就是包含[`encloses(Range)`]：如果内区间的边界没有超出外区间的边界，则外区间包含内区间。包含判断的结果完全取决于区间端点的比较！

- [3..6] 包含[4..5] ；
- (3..6) 包含(3..6) ；
- [3..6] 包含[4..4)，虽然后者是空区间；
- (3..6]不 包含[3..6] ；
- [4..5]不 包含(3..6)，虽然前者包含了后者的所有值，离散域[discrete domains]可以解决这个问题（见8.5节）；
- [3..6]不 包含(1..1]，虽然前者包含了后者的所有值。

包含是一种偏序关系[partial ordering]。基于包含关系的概念，Range还提供了以下运算方法。

相连[isConnected]

`Range.isConnected(Range)` 判断区间是否是相连的。具体来说，isConnected测试是否有区间同时包含于这两个区间，这等同于数学上的定义”两个区间的并集是连续集合的形式”（空区间的特殊情况除外）。

相连是一种自反的[reflexive]、对称的[symmetric]关系。

```
Range.closed(3, 5).isConnected(Range.open(5, 10)); // returns true
Range.closed(0, 9).isConnected(Range.closed(3, 4)); // returns true
Range.closed(0, 5).isConnected(Range.closed(3, 9)); // returns true
Range.open(3, 5).isConnected(Range.open(5, 10)); // returns false
Range.closed(1, 5).isConnected(Range.closed(6, 10)); // returns false
```

交集[intersection]

`Range.intersection(Range)` 返回两个区间的交集：既包含于第一个区间，又包含于另一个区间的最大区间。当且仅当两个区间是相连的，它们才有交集。如果两个区间没有交集，该方法将抛出IllegalArgumentException 。

交集是可互换的[commutative] 、关联的[associative] 运算[operation]。

```
Range.closed(3, 5).intersection(Range.open(5, 10)); // returns (5, 5]
Range.closed(0, 9).intersection(Range.closed(3, 4)); // returns [3, 4]
Range.closed(0, 5).intersection(Range.closed(3, 9)); // returns [3, 5]
Range.open(3, 5).intersection(Range.open(5, 10)); // throws IAE
Range.closed(1, 5).intersection(Range.closed(6, 10)); // throws IAE
```

跨区间[span]

`Range.span(Range)` 返回”同时包括两个区间的最小区间”，如果两个区间相连，那就是它们的并集。

span是可互换的[commutative] 、关联的[associative] 、闭合的[closed]运算[operation]。

```
Range.closed(3, 5).span(Range.open(5, 10)); // returns [3, 10)
Range.closed(0, 9).span(Range.closed(3, 4)); // returns [0, 9]
Range.closed(0, 5).span(Range.closed(3, 9)); // returns [0, 9]
Range.open(3, 5).span(Range.open(5, 10)); // returns (3, 10)
Range.closed(1, 5).span(Range.closed(6, 10)); // returns [1, 10]
```

离散域

部分（但不是全部）可比较类型是离散的，即区间的上下边界都是可枚举的。

在Guava中，用DiscreteDomain<C>实现类型C的离散形式操作。一个离散域总是代表某种类型值的全集；它不能代表类似"素数"、"长度为5的字符串"或"午夜的时间戳"这样的局部域。

DiscreteDomain提供的离散域实例包括：

类型	离散域
Integer	integers()
Long	longs()

一旦获取了DiscreteDomain实例，你就可以使用下面的Range运算方法：

- ContiguousSet.create(range, domain)：用ImmutableSortedSet<C>形式表示Range<C>中符合离散域定义的元素，并增加一些额外操作——译者注：实际返回***ImmutableSortedSet***的子类__*ContiguousSet*。（对无限区间不起作用，除非类型C本身是有限的，比如int就是可枚举的）
- canonical(domain)：把离散域转为区间的"规范形式"。如果ContiguousSet.create(a, domain).equals(ContiguousSet.create(b, domain))并且!a.isEmpty()，则有a.canonical(domain).equals(b.canonical(domain))。（这并不意味着a.equals(b)）

```
ImmutableSortedSet set = ContigousSet.create(Range.open(1, 5), i
screteDomain.integers());
//set包含[2, 3, 4]
ContiguousSet.create(Range.greaterThan(0), DiscreteDomain.intege
rs());
//set包含[1, 2, ..., Integer.MAX_VALUE]
```

注意，ContiguousSet.create并没有真的构造了整个集合，而是返回了set形式的区间视图。

你自己的离散域

你可以创建自己的离散域，但必须记住DiscreteDomain契约的几个重要方面。

- 一个离散域总是代表某种类型值的全集；它不能代表类似”素数”或”长度为5的字符串”这样的局部域。所以举例来说，你无法构造一个DiscreteDomain以表示精确到秒的JODA DateTime日期集合：因为那将无法包含JODA DateTime的所有值。
- DiscreteDomain可能是无限的——比如BigInteger DiscreteDomain。这种情况下，你应当用minValue()和maxValue()的默认实现，它们会抛出NoSuchElementException。但Guava禁止把无限区间传入ContiguousSet.create——译者注：那明显得不到一个可枚举的集合。

如果我需要一个Comparator呢？

我们想要在Range的可用性与API复杂性之间找到特定的平衡，这部分导致了我们没有提供基于Comparator的接口：我们不需要操心区间是怎样基于不同Comparator互动的；所有API签名都是简单明确的；这样更好。

另一方面，如果你需要任意Comparator，可以按下列其中一项来做：

- 使用通用的Predicate接口，而不是Range类。（Range实现了Predicate接口，因此可以用Predicates.compose(range, function)获取Predicate实例）
- 使用包装类以定义期望的排序。

译者注：实际上*Range*规定元素类型必须是*Comparable*，这已经满足了大多数需求。如果需要自定义特殊的比较逻辑，可以用*Predicates.compose(range, function)*组合比较的*function*。

9-I/O

原文链接 译文链接 译者：沈义扬

字节流和字符流

Guava使用术语"流" 来表示可关闭的，并且在底层资源中有位置状态的I/O数据流。术语"字节流"指的是InputStream或OutputStream，"字符流"指的是Reader 或Writer（虽然他们的接口Readable 和Appendable被更多地用于方法参数）。相应的工具方法分别在 ByteStreams 和 CharStreams 中。

大多数Guava流工具一次处理一个完整的流，并且/或者为了效率自己处理缓冲。还要注意到，接受流为参数的Guava方法不会关闭这个流：关闭流的职责通常属于打开流的代码块。

其中的一些工具方法列举如下：

ByteStreams	CharStreams
byte[] toByteArray(InputStream)	String toString(Readable)
N/A	List<String> readLi
long copy(InputStream, OutputStream)	long copy(Readable, Appen
void readFully(InputStream, byte[])	N/A
void skipFully(InputStream, long)	void skipFully(Reader, lo
OutputStream nullOutputStream()	Writer nullWriter()

关于**InputSupplier** 和**OutputSupplier**要注意：

在ByteStreams、CharStreams以及com.google.common.io包中的一些其他类中，某些方法仍然在使用InputSupplier和OutputSupplier接口。这两个借口和相关的方法是不推荐使用的：它们已经被下面描述的source和sink类型取代了，并且最终会被移除。

源与汇

通常我们都会创建I/O工具方法，这样可以避免在做基础运算时总是直接和流打交道。例如，Guava有Files.toByteArray(File) 和Files.write(File, byte[])。然而，流工具方法的创建经常最终导致散落各处的相似方法，每个方法读取不同类型的源

或写入不同类型的汇[sink]。例如，Guava中的Resources.toByteArray(URL)和Files.toByteArray(File)做了同样的事情，只不过数据源一个是URL，一个是文件。

为了解决这个问题，Guava有一系列关于源与汇的抽象。源或汇指某个你知道如何从中打开流的资源，比如File或URL。源是可读的，汇是可写的。此外，源与汇按照字节和字符划分类型。

字节	字符	
读	`ByteSource`	`CharSource`
写	`ByteSink`	`CharSink`

源与汇API的好处是它们提供了通用的一组操作。比如，一旦你把数据源包装成了ByteSource，无论它原先的类型是什么，你都得到了一组按字节操作的方法。

创建源与汇

Guava提供了若干源与汇的实现：

字节	字符
`Files.asByteSource(File)`	`Files.asCharSource(Fi`
`Files.asByteSink(File, FileWriteMode...)`	`Files.asCharSink(File`
`Resources.asByteSource(URL)`	`Resources.asCharSourc`
`ByteSource.wrap(byte[])`	`CharSource.wrap(CharS`
`ByteSource.concat(ByteSource...)`	`CharSource.concat(Cha`
`ByteSource.slice(long, long)`	N/A
N/A	`ByteSource.asCharSour`
N/A	`ByteSink.asCharSink(C`

此外，你也可以继承这些类，以创建新的实现。

注：把已经打开的流（比如InputStream）包装为源或汇听起来是很有诱惑力的，但是应该避免这样做。源与汇的实现应该在每次openStream()方法被调用时都创建一个新的流。始终创建新的流可以让源或汇管理流的整个生命周期，并且让多次调用openStream()返回的流都是可用的。此外，如果你在创建源或汇之前创建了流，你不得不在异常的时候自己保证关闭流，这压根就违背了发挥源与汇API优点的初衷。

使用源与汇

一旦有了源与汇的实例，就可以进行若干读写操作。

通用操作

所有源与汇都有一些方法用于打开新的流用于读或写。默认情况下，其他源与汇操作都是先用这些方法打开流，然后做一些读或写，最后保证流被正确地关闭了。这些方法列举如下：

- openStream()：根据源与汇的类型，返回InputStream、OutputStream、Reader或者Writer。
- openBufferedStream()：根据源与汇的类型，返回InputStream、OutputStream、BufferedReader或者BufferedWriter。返回的流保证在必要情况下做了缓冲。例如，从字节数组读数据的源就没有必要再在内存中作缓冲，这就是为什么该方法针对字节源不返回BufferedInputStream。字符源属于例外情况，它一定返回BufferedReader，因为BufferedReader中才有readLine()方法。

源操作

字节源	字符源
byte[] read()	String read()
N/A	ImmutableList<String>
N/A	String readFirstLine()
long copyTo(ByteSink)	long copyTo(CharSink)
long copyTo(OutputStream)	long copyTo(Appendable)
long size() (in bytes)	N/A
boolean isEmpty()	boolean isEmpty()
boolean contentEquals(ByteSource)	N/A
HashCode hash(HashFunction)	N/A

汇操作

字节汇	字符汇
void write(byte[])	void write(CharSequence)
long writeFrom(InputStream)	long writeFrom(Readable)
N/A	void writeLines(Iterable<? exte
N/A	void writeLines(Iterable<? exte

范例

```
//Read the lines of a UTF-8 text file
ImmutableList<String> lines = Files.asCharSource(file, Charsets.
UTF_8).readLines();
//Count distinct word occurrences in a file
Multiset<String> wordOccurrences = HashMultiset.create(
        Splitter.on(CharMatcher.WHITESPACE)
            .trimResults()
            .omitEmptyStrings()
            .split(Files.asCharSource(file, Charsets.UTF_8).read
()));

//SHA-1 a file
HashCode hash = Files.asByteSource(file).hash(Hashing.sha1());

//Copy the data from a URL to a file
Resources.asByteSource(url).copyTo(Files.asByteSink(file));
```

文件操作

除了创建文件源和文件的方法，Files类还包含了若干你可能感兴趣的便利方法。

`createParentDirs(File)`	必要时为文件创建父目录
`getFileExtension(String)`	返回给定路径所表示文件的扩展名
`getNameWithoutExtension(String)`	返回去除了扩展名的文件名
`simplifyPath(String)`	规范文件路径，并不总是与文件系统一致，请仔细测试
`fileTreeTraverser()`	返回TreeTraverser用于遍历文件树

10-散列

原文链接 译文链接 译者：沈义扬

概述

Java内建的散列码[hash code]概念被限制为32位，并且没有分离散列算法和它们所作用的数据，因此很难用备选算法进行替换。此外，使用Java内建方法实现的散列码通常是劣质的，部分是因为它们最终都依赖于JDK类中已有的劣质散列码。

Object.hashCode往往很快，但是在预防碰撞上却很弱，也没有对分散性的预期。这使得它们很适合在散列表中运用，因为额外碰撞只会带来轻微的性能损失，同时差劲的分散性也可以容易地通过再散列来纠正（Java中所有合理的散列表都用了再散列方法）。然而，在简单散列表以外的散列运用中，Object.hashCode几乎总是达不到要求——因此，有了com.google.common.hash包。

散列包的组成

在这个包的Java doc中，我们可以看到很多不同的类，但是文档中没有明显地表明它们是怎样 一起配合工作的。在介绍散列包中的类之前，让我们先来看下面这段代码范例：

```
HashFunction hf = Hashing.md5();
HashCode hc = hf.newHasher()
        .putLong(id)
        .putString(name, Charsets.UTF_8)
        .putObject(person, personFunnel)
        .hash();
```

HashFunction

HashFunction是一个单纯的（引用透明的）、无状态的方法，它把任意的数据块映射到固定数目的位指，并且保证相同的输入一定产生相同的输出，不同的输入尽可能产生不同的输出。

Hasher

HashFunction的实例可以提供有状态的Hasher，Hasher提供了流畅的语法把数据添加到散列运算，然后获取散列值。Hasher可以接受所有原生类型、字节数组、字节数组的片段、字符序列、特定字符集的字符序列等等，或者任何给定了Funnel实现的对象。

Hasher实现了PrimitiveSink接口，这个接口为接受原生类型流的对象定义了fluent风格的API

Funnel

Funnel描述了如何把一个具体的对象类型分解为原生字段值，从而写入PrimitiveSink。比如，如果我们有这样一个类：

```
class Person {
    final int id;
    final String firstName;
    final String lastName;
    final int birthYear;
}
```

它对应的Funnel实现可能是：

```
Funnel<Person> personFunnel = new Funnel<Person>() {
    @Override
    public void funnel(Person person, PrimitiveSink into) {
        into
            .putInt(person.id)
            .putString(person.firstName, Charsets.UTF_8)
            .putString(person.lastName, Charsets.UTF_8)
            .putInt(birthYear);
    }
}
```

注：putString("abc", Charsets.UTF_8).putString("def", Charsets.UTF_8)完全等同于putString("ab", Charsets.UTF_8).putString("cdef", Charsets.UTF_8)，因为它们提供了相同的字节序列。这可能带来预料之外的散列冲突。增加某种形式的分隔符有助于消除散列冲突。

HashCode

一旦Hasher被赋予了所有输入，就可以通过hash()方法获取HashCode实例（多次调用hash()方法的结果是不确定的）。HashCode可以通过asInt()、asLong()、asBytes()方法来做相等性检测，此外，writeBytesTo(array, offset, maxLength)把散列值的前maxLength字节写入字节数组。

布鲁姆过滤器[BloomFilter]

布鲁姆过滤器是哈希运算的一项优雅运用，它可以简单地基于Object.hashCode()实现。简而言之，布鲁姆过滤器是一种概率数据结构，它允许你检测某个对象是一定不在过滤器中，还是可能已经添加到过滤器了。布鲁姆过滤器的维基页面对此作了

全面的介绍，同时我们推荐github中的一个教程。

Guava散列包有一个内建的布鲁姆过滤器实现，你只要提供Funnel就可以使用它。你可以使用create(Funnel funnel, int expectedInsertions, double falsePositiveProbability)方法获取BloomFilter<T>，缺省误检率[falsePositiveProbability]为3%。BloomFilter<T>提供了boolean mightContain(T) 和void put(T)，它们的含义都不言自明了。

```
BloomFilter<Person> friends = BloomFilter.create(personFunnel, 5
00, 0.01);
for(Person friend : friendsList) {
    friends.put(friend);
}

// 很久以后
if (friends.mightContain(dude)) {
    //dude不是朋友还运行到这里的概率为1%
    //在这儿，我们可以在做进一步精确检查的同时触发一些异步加载
}
```

Hashing类

Hashing类提供了若干散列函数，以及运算HashCode对象的工具方法。

已提供的散列函数

`md5()`	`murmur3_128()`	`murmur3_32()`	`sha1()`
`sha256()`	`sha512()`	`goodFastHash(int bits)`	

HashCode运算

方法	描述
`HashCode combineOrdered( Iterable<HashCode>)`	以有序方式联接散列码，如果两个散列集合用该方法联接出的散列码相同，那么散列集合的元素可能是顺序相等的
`HashCode combineUnordered( Iterable<HashCode>)`	以无序方式联接散列码，如果两个散列集合用该方法联接出的散列码相同，那么散列集合的元素可能在某种排序下是相等的
`int consistentHash( HashCode, int buckets)`	为给定的"桶"大小返回一致性哈希值。当"桶"增长时，该方法保证最小程度的一致性哈希值变化。详见一致性哈希。

11-事件总线

原文链接 译文连接 译者：沈义扬

传统上，Java的进程内事件分发都是通过发布者和订阅者之间的显式注册实现的。设计EventBus就是为了取代这种显示注册方式，使组件间有了更好的解耦。EventBus不是通用型的发布-订阅实现，不适用于进程间通信。

范例

```
// Class is typically registered by the container.
class EventBusChangeRecorder {
    @Subscribe public void recordCustomerChange(ChangeEvent e) {
        recordChange(e.getChange());
    }
}
// somewhere during initialization
eventBus.register(new EventBusChangeRecorder());
// much later
public void changeCustomer() {
    ChangeEvent event = getChangeEvent();
    eventBus.post(event);
}
```

一分钟指南

把已有的进程内事件分发系统迁移到EventBus非常简单。

事件监听者[Listeners]

监听特定事件（如，CustomerChangeEvent）：

- 传统实现：定义相应的事件监听者类，如CustomerChangeEventListener；
- EventBus实现：以CustomerChangeEvent为唯一参数创建方法，并用Subscribe注解标记。

把事件监听者注册到事件生产者：

- 传统实现：调用事件生产者的registerCustomerChangeEventListener方法；这些方法很少定义在公共接口中，因此开发者必须知道所有事件生产者的类型，才能正确地注册监听者；
- EventBus实现：在EventBus实例上调用EventBus.register(Object)方法；请保证事件生产者和监听者共享相同的EventBus实例。

按事件超类监听（如，EventObject甚至Object）：

- 传统实现：很困难，需要开发者自己去实现匹配逻辑；
- EventBus实现：EventBus自动把事件分发给事件超类的监听者，并且允许监听者声明监听接口类型和泛型的通配符类型（wildcard，如？super XXX）。

检测没有监听者的事件：

- 传统实现：在每个事件分发方法中添加逻辑代码（也可能适用AOP）；
- EventBus实现：监听DeadEvent；EventBus会把所有发布后没有监听者处理的事件包装为DeadEvent（对调试很便利）。

事件生产者[Producers]

管理和追踪监听者：

- 传统实现：用列表管理监听者，还要考虑线程同步；或者使用工具类，如EventListenerList；
- EventBus实现：EventBus内部已经实现了监听者管理。

向监听者分发事件：

- 传统实现：开发者自己写代码，包括事件类型匹配、异常处理、异步分发；
- EventBus实现：把事件传递给 EventBus.post(Object)方法。异步分发可以直接用EventBus的子类AsyncEventBus。

术语表

事件总线系统使用以下术语描述事件分发：

事件	可以向事件总线发布的对象
订阅	向事件总线注册监听者以接受事件的行为
监听者	提供一个处理方法，希望接受和处理事件的对象
处理方法	监听者提供的公共方法，事件总线使用该方法向监听者发送事件；该方法应该用Subscribe注解
发布消息	通过事件总线向所有匹配的监听者提供事件

常见问题解答[FAQ]

为什么一定要创建**EventBus**实例，而不是使用单例模式？

EventBus不想给定开发者怎么使用；你可以在应用程序中按照不同的组件、上下文或业务主题分别使用不同的事件总线。这样的话，在测试过程中开启和关闭某个部分的事件总线，也会变得更简单，影响范围更小。

当然，如果你想在进程范围内使用唯一的事件总线，你也可以自己这么做。比如在容器中声明EventBus为全局单例，或者用一个静态字段存放EventBus，如果你喜欢的话。

简而言之，EventBus不是单例模式，是因为我们不想为你做这个决定。你喜欢怎么用就怎么用吧。

可以从事件总线中注销监听者吗？

当然可以，使用EventBus.unregister(Object)方法，但我们发现这种需求很少：

- 大多数监听者都是在启动或者模块懒加载时注册的，并且在应用程序的整个生命周期都存在；
- 可以使用特定作用域的事件总线来处理临时事件，而不是注册/注销监听者；比如在请求作用域[request-scoped]的对象间分发消息，就可以同样适用请求作用域的事件总线；
- 销毁和重建事件总线的成本很低，有时候可以通过销毁和重建事件总线来更改分发规则。

为什么使用注解标记处理方法，而不是要求监听者实现接口？

我们觉得注解和实现接口一样传达了明确的语义，甚至可能更好。同时，使用注解也允许你把处理方法放到任何地方，和使用业务意图清晰的方法命名。

传统的Java实现中，监听者使用方法很少的接口——通常只有一个方法。这样做有一些缺点:

- 监听者类对给定事件类型，只能有单一处理逻辑；
- 监听者接口方法可能冲突；
- 方法命名只和事件相关（handleChangeEvent），不能表达意图（recordChangeInJournal）；
- 事件通常有自己的接口，而没有按类型定义的公共父接口（如所有的UI事件接口）。

接口实现监听者的方式很难做到简洁，这甚至引出了一个模式，尤其是在Swing应用中，那就是用匿名类实现事件监听者的接口。比较以下两种实现：

```
class ChangeRecorder {
    void setCustomer(Customer cust) {
        cust.addChangeListener(new ChangeListener() {
            public void customerChanged(ChangeEvent e) {
                recordChange(e.getChange());
            }
        };
    }
}
```

```
//这个监听者类通常由容器注册给事件总线
class EventBusChangeRecorder {
    @Subscribe public void recordCustomerChange(ChangeEvent e) {
        recordChange(e.getChange());
    }
}
```

第二种实现的业务意图明显更加清晰：没有多余的代码，并且处理方法的名字是清晰和有意义的。

通用的监听者接口**Handler<T>**怎么样？

有些人已经建议过用泛型定义一个通用的监听者接口Handler<T>。这有点牵扯到Java类型擦除的问题，假设我们有如下这个接口：

```
interface Handler<T> {
    void handleEvent(T event);
}
```

因为类型擦除，Java禁止一个类使用不同的类型参数多次实现同一个泛型接口（即不可能出现MultiHandler implements Handler<Type1>, Handler<Type2>）。这比起传统的Java事件机制也是巨大的退步，至少传统的Java Swing监听者接口使用了不同的方法把不同的事件区分开。

EventBus不是破坏了静态类型，排斥了自动重构支持吗？**

有些人被EventBus的register(Object) 和post(Object)方法直接使用Object做参数吓坏了。

这里使用Object参数有一个很好的理由：EventBus对事件监听者类型和事件本身的类型都不作任何限制。

另一方面，处理方法必须要明确地声明参数类型——期望的事件类型（或事件的父类型）。因此，搜索一个事件的类型引用，可以马上找到针对该事件的处理方法，对事件类型的重命名也会在IDE中自动更新所有的处理方法。

在EventBus的架构下，你可以任意重命名@Subscribe注解的处理方法，并且这类重命名不会被传播（即不会引起其他类的修改），因为对EventBus来说，处理方法的名字是无关紧要的。如果测试代码中直接调用了处理方法，那么当然，重命名处理方法会引起测试代码的变动，但使用EventBus触发处理方法的代码就不会发生变更。我们认为这是EventBus的特性，而不是漏洞：能够任意重命名处理方法，可以让你的处理方法命名更清晰。

如果我注册了一个没有任何处理方法的监听者，会发生什么？

什么也不会发生。

EventBus旨在与容器和模块系统整合，Guice就是个典型的例子。在这种情况下，可以方便地让容器/工厂/运行环境传递任意创建好的对象给EventBus的register(Object)方法。

这样，任何容器/工厂/运行环境创建的对象都可以简便地通过暴露处理方法挂载到系统的事件模块。

编译时能检测到**EventBus**的哪些问题？

Java类型系统可以明白地检测到的任何问题。比如，为一个不存在的事件类型定义处理方法。

运行时往**EventBus**注册监听者，可以立即检测到哪些问题？

一旦调用了register(Object) 方法，EventBus就会检查监听者中的处理方法是否结构正确的[well-formedness]。具体来说，就是每个用@Subscribe注解的方法都只能有一个参数。

违反这条规则将引起IllegalArgumentException（这条规则检测也可以用APT在编译时完成，不过我们还在研究中）。

哪些问题只能在之后事件传播的运行时才会被检测到？

如果组件传播了一个事件，但找不到相应的处理方法，EventBus可能会指出一个错误（通常是指出@Subscribe注解的缺失，或没有加载监听者组件）。

请注意这个指示并不一定表示应用有问题。一个应用中可能有好多场景会故意忽略某个事件，尤其当事件来源于不可控代码时

你可以注册一个处理方法专门处理DeadEvent类型的事件。每当EventBus收到没有对应处理方法的事件，它都会将其转化为DeadEvent，并且传递给你注册的DeadEvent处理方法——你可以选择记录或修复该事件。

怎么测试监听者和它们的处理方法？

因为监听者的处理方法都是普通方法，你可以简便地在测试代码中模拟EventBus调用这些方法。

为什么我不能在**EventBus上使用<泛型魔法>？**

EventBus旨在很好地处理一大类用例。我们更喜欢针对大多数用例直击要害，而不是在所有用例上都保持体面。

此外，泛型也让EventBus的可扩展性——让它有益、高效地扩展，同时我们对EventBus的增补不会和你们的扩展相冲突——成为一个非常棘手的问题。

如果你真的很想用泛型，EventBus目前还不能提供，你可以提交一个问题并且设计自己的替代方案。

12-数学运算

原文链接 译文链接 译者：沈义扬

范例

```
int logFloor = LongMath.log2(n, FLOOR);
int mustNotOverflow = IntMath.checkedMultiply(x, y);
long quotient = LongMath.divide(knownMultipleOfThree, 3, Roundin
gMode.UNNECESSARY); // fail fast on non-multiple of 3
BigInteger nearestInteger = DoubleMath.roundToBigInteger(d, Roun
dingMode.HALF_EVEN);
BigInteger sideLength = BigIntegerMath.sqrt(area, CEILING);
```

为什么使用Guava Math

- Guava Math针对各种不常见的溢出情况都有充分的测试；对溢出语义，Guava文档也有相应的说明；如果运算的溢出检查不能通过，将导致快速失败；
- Guava Math的性能经过了精心的设计和调优；虽然性能不可避免地依据具体硬件细节而有所差异，但Guava Math的速度通常可以与Apache Commons的MathUtils相比，在某些场景下甚至还有显著提升；
- Guava Math在设计上考虑了可读性和正确的编程习惯；IntMath.log2(x, CEILING) 所表达的含义，即使在快速阅读时也是清晰明确的。而32-Integer.numberOfLeadingZeros(x – 1)对于阅读者来说则不够清晰。

注意：*Guava Math*和*GWT*格外不兼容，这是因为*Java*和*Java Script*语言的运算溢出逻辑不一样。

整数运算

Guava Math主要处理三种整数类型：int、long和BigInteger。这三种类型的运算工具类分别叫做IntMath、LongMath和BigIntegerMath。

有溢出检查的运算

Guava Math提供了若干有溢出检查的运算方法：结果溢出时，这些方法将快速失败而不是忽略溢出

`IntMath.checkedAdd`	`LongMath.checkedAdd`
`IntMath.checkedSubtract`	`LongMath.checkedSubtract`
`IntMath.checkedMultiply`	`LongMath.checkedMultiply`
`IntMath.checkedPow`	`LongMath.checkedPow`

```
IntMath.checkedAdd(Integer.MAX_VALUE, Integer.MAX_VALUE); // thr
ows ArithmeticException
```

实数运算

IntMath、LongMath和BigIntegerMath提供了很多实数运算的方法，并把最终运算结果舍入成整数。这些方法接受一个java.math.RoundingMode枚举值作为舍入的模式：

- DOWN：向零方向舍入（去尾法）
- UP：远离零方向舍入
- FLOOR：向负无限大方向舍入
- CEILING：向正无限大方向舍入
- UNNECESSARY：不需要舍入，如果用此模式进行舍入，应直接抛出ArithmeticException
- HALF_UP：向最近的整数舍入，其中x.5远离零方向舍入
- HALF_DOWN：向最近的整数舍入，其中x.5向零方向舍入
- HALF_EVEN：向最近的整数舍入，其中x.5向相邻的偶数舍入

这些方法旨在提高代码的可读性，例如，divide(x, 3, CEILING) 即使在快速阅读时也是清晰。此外，这些方法内部采用构建整数近似值再计算的实现，除了在构建sqrt（平方根）运算的初始近似值时有浮点运算，其他方法的运算全过程都是整数或位运算，因此性能上更好。

运算	IntMath	LongMath
除法	`divide(int, int, RoundingMode)`	`divide(long, long, Roundin`
2为底的对数	`log2(int, RoundingMode)`	`log2(long, RoundingMode)`
10为底的对数	`log10(int, RoundingMode)`	`log10(long, RoundingMode)`
平方根	`sqrt(int, RoundingMode)`	`sqrt(long, RoundingMode)`

```
// returns 31622776601683793319988935444327185337195551393252
BigIntegerMath.sqrt(BigInteger.TEN.pow(99), RoundingMode.HALF_EVEN);
```

附加功能

Guava还另外提供了一些有用的运算函数

运算	IntMath	LongMath	BigIntegerMath*
最大公约数	`gcd(int, int)`	`gcd(long, long)`	`BigInteger.gcd(B`
取模	`mod(int, int)`	`mod(long, long)`	`BigInteger.mod(B`
取幂	`pow(int, int)`	`pow(long, int)`	`BigInteger.pow(i`
是否2的幂	`isPowerOfTwo(int)`	`isPowerOfTwo(long)`	`isPowerOfTwo(Big`
阶乘*	`factorial(int)`	`factorial(int)`	`factorial(int)`
二项式系数*	`binomial(int, int)`	`binomial(int, int)`	`binomial(int, in`

*BigInteger的最大公约数和取模运算由JDK提供

*阶乘和二项式系数的运算结果如果溢出，则返回MAX_VALUE

浮点数运算

JDK比较彻底地涵盖了浮点数运算，但Guava在DoubleMath类中也提供了一些有用的方法。

`isMathematicalInteger(double)`	判断该浮点数是不是一个整数
`roundToInt(double, RoundingMode)`	舍入为int；对无限小数、溢出抛出异常
`roundToLong(double, RoundingMode)`	舍入为long；对无限小数、溢出抛出异常
`roundToBigInteger(double, RoundingMode)`	舍入为BigInteger；对无限小数抛出异常
`log2(double, RoundingMode)`	2的浮点对数，并且舍入为int，比JDK的Math.log(double) 更快

13-反射

译者：万天慧(武祖)

由于类型擦除，你不能够在运行时传递泛型类对象——你可能想强制转换它们，并假装这些对象是有泛型的，但实际上它们没有。

举个例子：

```
ArrayList<String> stringList = Lists.newArrayList();
ArrayList<Integer> intList = Lists.newArrayList();
System.out.println(stringList.getClass().isAssignableFrom(intLis
t.getClass()));
returns true, even though ArrayList<String> is not assignable fr
om ArrayList<Integer>
```

Guava提供了TypeToken, 它使用了基于反射的技巧甚至让你在运行时都能够巧妙的操作和查询泛型类型。想象一下TypeToken是创建，操作，查询泛型类型（以及，隐含的类）对象的方法。

Guice用户特别注意：TypeToken与类Guice的TypeLiteral很相似，但是有一个点特别不同：它能够支持非具体化的类型，例如T，List<T>，甚至是List<? extends Number>；TypeLiteral则不能支持。TypeToken也能支持序列化并且提供了很多额外的工具方法。

背景：类型擦除与反射

Java不能在运行时保留对象的泛型类型信息。如果你在运行时有一个ArrayList<String>对象，你不能够判定这个对象是有泛型类型ArrayList<String>的—— 并且通过不安全的原始类型，你可以将这个对象强制转换成ArrayList<Object>。

但是，反射允许你去检测方法和类的泛型类型。如果你实现了一个返回List的方法，并且你用反射获得了这个方法的返回类型，你会获得代表List<String>的ParameterizedType。

TypeToken类使用这种变通的方法以最小的语法开销去支持泛型类型的操作。

介绍

获取一个基本的、原始类的TypeToken非常简单：

```
TypeToken<String> stringTok = TypeToken.of(String.class);
TypeToken<Integer> intTok = TypeToken.of(Integer.class);
```

为获得一个含有泛型的类型的TypeToken —— 当你知道在编译时的泛型参数类型 —— 你使用一个空的匿名内部类：

```
TypeToken<List<String>> stringListTok = new TypeToken<List<Strin
g>>() {};
```

或者你想故意指向一个通配符类型：

```
TypeToken<Map<?, ?>> wildMapTok = new TypeToken<Map<?, ?>>() {};
```

TypeToken提供了一种方法来动态的解决泛型类型参数，如下所示：

```
static <K, V> TypeToken<Map<K, V>> mapToken(TypeToken<K> keyToke
n, TypeToken<V> valueToken) {
    return new TypeToken<Map<K, V>>() {}
        .where(new TypeParameter<K>() {}, keyToken)
        .where(new TypeParameter<V>() {}, valueToken);
}
...
TypeToken<Map<String, BigInteger>> mapToken = mapToken(
    TypeToken.of(String.class),
    TypeToken.of(BigInteger.class)
);
TypeToken<Map<Integer, Queue<String>>> complexToken = mapToken(
   TypeToken.of(Integer.class),
   new TypeToken<Queue<String>>() {}
);
```

注意如果mapToken只是返回了new TypeToken>()，它实际上不能把具体化的类型分配到K和V上面，举个例子

```
class Util {
    static <K, V> TypeToken<Map<K, V>> incorrectMapToken() {
        return new TypeToken<Map<K, V>>() {};
    }
}
System.out.println(Util.<String, BigInteger>incorrectMapToken())
;
// just prints out "java.util.Map<K, V>"
```

或者，你可以通过一个子类（通常是匿名）来捕获一个泛型类型并且这个子类也可以用来替换知道参数类型的上下文类。

```
abstract class IKnowMyType<T> {
    TypeToken<T> type = new TypeToken<T>(getClass()) {};
}
...
new IKnowMyType<String>() {}.type; // returns a correct TypeToke
n<String>
```

使用这种技术，你可以，例如，获得知道他们的元素类型的类。

查询

TypeToken支持很多种类能支持的查询，但是也会把通用的查询约束考虑在内。

支持的查询操作包括：

方法	描述
getType()	获得包装的java.lang.reflect.Type.
getRawType()	返回大家熟知的运行时类
getSubtype(Class<?>)	返回那些有特定原始类的子类型。举个例子，如果这有一个Iterable并且参数是List.class，那么返回将是List。
getSupertype(Class<?>)	产生这个类型的超类，这个超类是指定的原始类型。举个例子，如果这是一个Set并且参数是Iterable.class，结果将会是Iterable。
isAssignableFrom(type)	如果这个类型是 assignable from 指定的类型，并且考虑泛型参数，返回true。List<? extends Number>是assignable from List，但List没有.
getTypes()	返回一个Set，包含了这个所有接口，子类和类是这个类型的类。返回的Set同样提供了classes()和interfaces()方法允许你只浏览超类和接口类。
isArray()	检查某个类型是不是数组，甚至是<? extends A[]>。
getComponentType()	返回组件类型数组。

resolveType

resolveType是一个可以用来“替代”context token（译者：不知道怎么翻译，只好去stackoverflow去问了）中的类型参数的一个强大而复杂的查询操作。例如，

```
TypeToken<Function<Integer, String>> funToken = new TypeToken<Function<Integer, String>>() {};

TypeToken<?> funResultToken = funToken.resolveType(Function.class.getTypeParameters()[1]));
// returns a TypeToken<String>
```

TypeToken将Java提供的TypeVariables和context token中的类型变量统一起来。这可以被用来一般性地推断出在一个类型相关方法的返回类型：

```
TypeToken<Map<String, Integer>> mapToken = new TypeToken<Map<String, Integer>>() {};
TypeToken<?> entrySetToken = mapToken.resolveType(Map.class.getMethod("entrySet").getGenericReturnType());
// returns a TypeToken<Set<Map.Entry<String, Integer>>>
```

Invokable

Guava的Invokable是对java.lang.reflect.Method和java.lang.reflect.Constructor的流式包装。它简化了常见的反射代码的使用。一些使用例子：

方法是否是public的?

JDK:

```
Modifier.isPublic(method.getModifiers())
```

Invokable:

```
invokable.isPublic()
```

方法是否是package private?

JDK:

```
!(Modifier.isPrivate(method.getModifiers()) || Modifier.isPublic(method.getModifiers()))
```

Invokable:

```
invokable.isPackagePrivate()
```

方法是否能够被子类重写？

JDK:

```
!(Modifier.isFinal(method.getModifiers())
|| Modifiers.isPrivate(method.getModifiers())
|| Modifiers.isStatic(method.getModifiers())
|| Modifiers.isFinal(method.getDeclaringClass().getModifiers()))
```

Invokable:

```
invokable.isOverridable()
```

方法的第一个参数是否被定义了注解@Nullable？

JDK:

```
for (Annotation annotation : method.getParameterAnnotations[0])
{
    if (annotation instanceof Nullable) {
        return true;
    }
}
return false;
```

Invokable:

```
invokable.getParameters().get(0).isAnnotationPresent(Nullable.cl
ass)
```

构造函数和工厂方法如何共享同样的代码？

你是否很想重复自己，因为你的反射代码需要以相同的方式工作在构造函数和工厂方法中？

Invokable提供了一个抽象的概念。下面的代码适合任何一种方法或构造函数：

```
invokable.isPublic();
invokable.getParameters();
invokable.invoke(object, args);
```

List的List.get(int)返回类型是什么？ Invokable提供了与众不同的类型解决方案：

```
Invokable<List<String>, ?> invokable = new TypeToken<List<String
>>()        {}.method(getMethod);
invokable.getReturnType(); // String.class
```

Dynamic Proxies

newProxy()

实用方法Reflection.newProxy(Class, InvocationHandler)是一种更安全，更方便的API，它只有一个单一的接口类型需要被代理来创建Java动态代理时

JDK:

```
Foo foo = (Foo) Proxy.newProxyInstance(
Foo.class.getClassLoader(),
new Class<?>[] {Foo.class},
invocationHandler);
```

Guava:

```
Foo foo = Reflection.newProxy(Foo.class, invocationHandler);
```

AbstractInvocationHandler

有时候你可能想动态代理能够更直观的支持equals()，hashCode()和toString()，那就是：

1. 一个代理实例equal另外一个代理实例，只要他们有同样的接口类型和equal的invocation handlers。
2. 一个代理实例的toString()会被代理到invocation handler的toString()，这样更容易自定义。

AbstractInvocationHandler实现了以上逻辑。

除此之外，AbstractInvocationHandler确保传递给handleInvocation(Object, Method, Object[])的参数数组永远不会空，从而减少了空指针异常的机会。

ClassPath

严格来讲，Java没有平台无关的方式来浏览类和类资源。不过一定的包或者工程下，还是能够实现的，比方说，去检查某个特定的工程的惯例或者某种一直遵从的约束。

ClassPath是一种实用工具，它提供尽最大努力的类路径扫描。用法很简单：

```
ClassPath classpath = ClassPath.from(classloader); // scans the
class path used by classloader
for (ClassPath.ClassInfo classInfo : classpath.getTopLevelClasse
s("com.mycomp.mypackage")) {
  ...
}
```

在上面的例子中，ClassInfo是被加载类的句柄。它允许程序员去检查类的名字和包的名字，让类直到需要的时候才被加载。

值得注意的是，ClassPath是一个尽力而为的工具。它只扫描jar文件中或者某个文件目录下的class文件。也不能扫描非URLClassLoader的自定义class loader管理的class，所以不要将它用于关键任务生产任务。

Class Loading

工具方法Reflection.initialize(Class…)能够确保特定的类被初始化——执行任何静态初始化。

使用这种方法的是一个代码异味，因为静态伤害系统的可维护性和可测试性。在有些情况下，你别无选择，而与传统的框架，操作间，这一方法有助于保持代码不那么丑。

JFreeChart 1.0.6 开发者指南

草稿（0.9.0）

译者：乖乖兔

2007-10-25

1 简介

1.1 什么是JFreeChart

1.1.1 概述

JFreeChart是一款免费的java图形开发类库。主要用来在application/ applets/ servlets/ jsp/上生成各种图表。JFreeChart是完全开源，并且严格遵循GNU的通用公共许可证，力保JFreeChart用户对源代码的自由修改与使用。

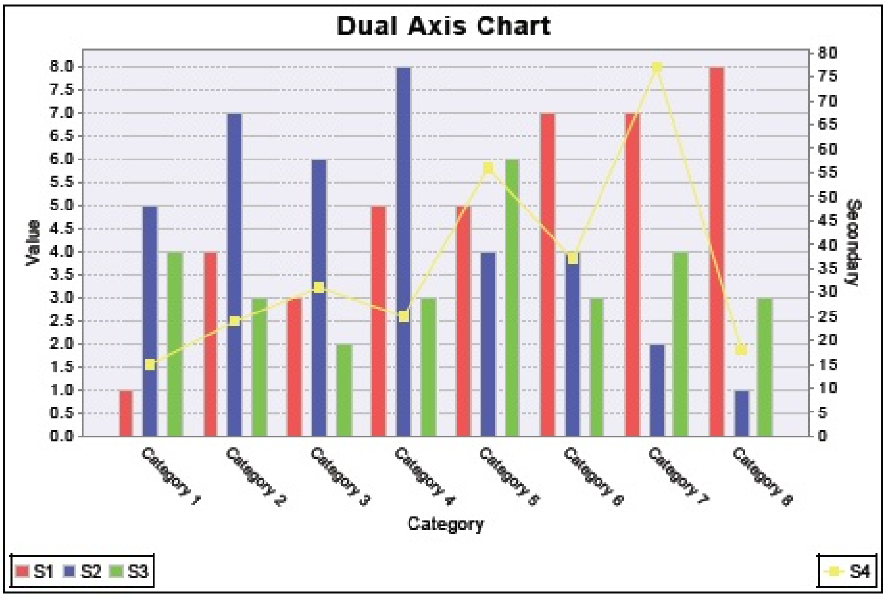


图1.1 一个简单的图表

图1.1就是一个典型的使用JFreeChart创建的图表。在本文后续章节将陆续展示更多的实例。

1.1.2 特征

JFreeChart能产生饼图（pie）、柱状/条形统计图（bar）、折线图（line）、散点图（scatter plots）、时序图（time series）、甘特图（Gantt）、仪表盘图（meter，比如刻度盘、温度计、罗盘等）、混合图、symbol图和风力方向图等。

主要特征如下：

- 定义接口的任何实现通俗易懂
- 易于导出PNG和JPEG图像文件格式（也可以使用java的图像I/O类库生成类库支持的任何格式）。
- 使用Graphics2D工具导出其他格式：
- 使用iText工具导出PDF格式文件
- 使用Batik工具导出SVG格式文件
- 图像工具栏
- 图表支持鼠标事件
- 支持注解。

- 产生HTML图像映射
- 可以工作于application/servlets/jsp/applets等环境。
- 完全开源、严格遵守GNU的通用公共认证协议。

JFreeChart完全由java语言编写，可以运行在java2的任何平台上（JDK1.3.1版本或者更高版本）

1.1.3 下载主页

JFreeChart 可以在下面的链接中找到：

http://www.jfree.org/jfreechart/ 这里我们可以找到JFreechart最新的版本，目前是1.0.6。包括图表实例、下载链接、javadoc文档、讨论社区等。

1.2 使用文档

文档有两个有效的版本：

- 免费版本 可以充JFreeChart主网站上下载免费版本《JFreeChart Installation Guide》，主要讲述内容是：JFreeChart的安装过程和JFreeChart实例的运行。
- 收费版本 需要支付一定费用才能获得《JFreeChart Developer Guide》，主要包括开发指南章节和JFreeChart类参考文档。

1.3 感谢

JFreeChart的代码和思路源于很多人。在这里我将感谢下面帮助JFreeChart成长的人,也许有些人员名字漏掉，望给予指正，在此表示歉意。名单如下：

Richard Atkinson, David Berry, Anthony Boulestreau, Jeremy Bowman, Daniel Bridenbecker, Nicolas Brodu, David Browning, Søren Caspersen, Chuanhao Chiu, Pascal Collet, Martin Cordova, Paolo Cova, Michael Duffy, Jonathan Gabbai, Serge V. Grachov, Hans-Jurgen Greiner, Joao Guilherme Del Valle, Aiman Han, Jon Iles, Wolfgang Irler, Xun Kang, Bill Kelemen, Norbert Kiesel, Gideon Krause, Arnaud Lelievre, David Li, Tin Luu, Craig MacFarlane, Achilleus Mantzios, Thomas Meier, Aaron Metzger, Jim Moore, Jonathan Nash, Barak Naveh, David M. O'Donnell, Krzysztof Paz, Tomer Peretz, Andrzej Porebski, Luke Quinane, Viktor Rajewski, Eduardo Ramalho, Michael Rauch, Cameron Riley, Dan Rivett, Michel Santos, Thierry Saura, Andreas Schneider, Jean-Luc Schwab, Bryan Scott, Roger Studner, Irv Thomae, Eric Thomas, Rich Unger, Daniel van Enckevort, Laurence Vanhelsuw´e, Sylvain Vieujot, JelaiWang, MarkWatson, Alex Weber, Matthew Wright, Christian W. Zuckschwerdt, Hari and Sam (oldman).

1.4 建议

如果您对本文档有任何的建议或想法，请发送：david.gilbert@object-refinery.com。

2 图表实例

2.1 介绍

本章节显示了许多使用JFreeChart创建的图表实例。内容特意对JFreeChart产生的图表类型做了概述。运行实例命令如下：

```
java -jar jfreechart-1.0.6-demo.jar
```

如果您购买了《JFreeChart Developer Guide》，可获得该实例的源代码。

2.2 饼图（Pie Charts）

JFreeChart能够创使用符合PieDataset接口标准的数据创建饼图。下图2.1显示了一个简单的饼图。

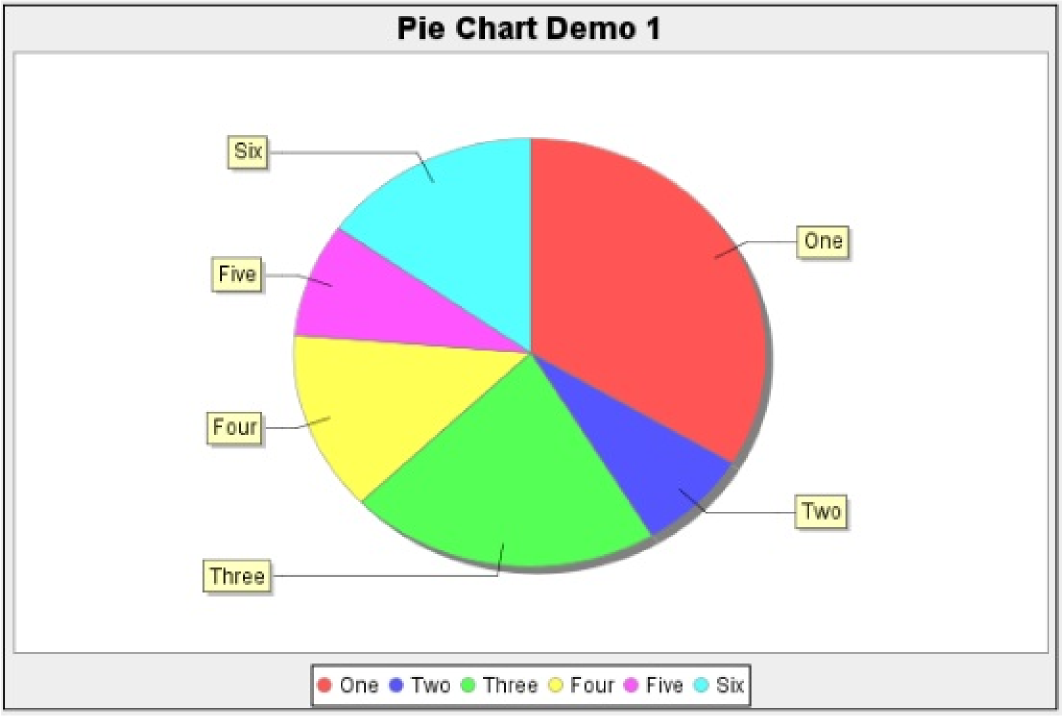


图2.1 一个简单的饼图（参见：PieChartDemo1.java）

其中 ，单个的区域也可以被“取出”，如下图2.2所示：

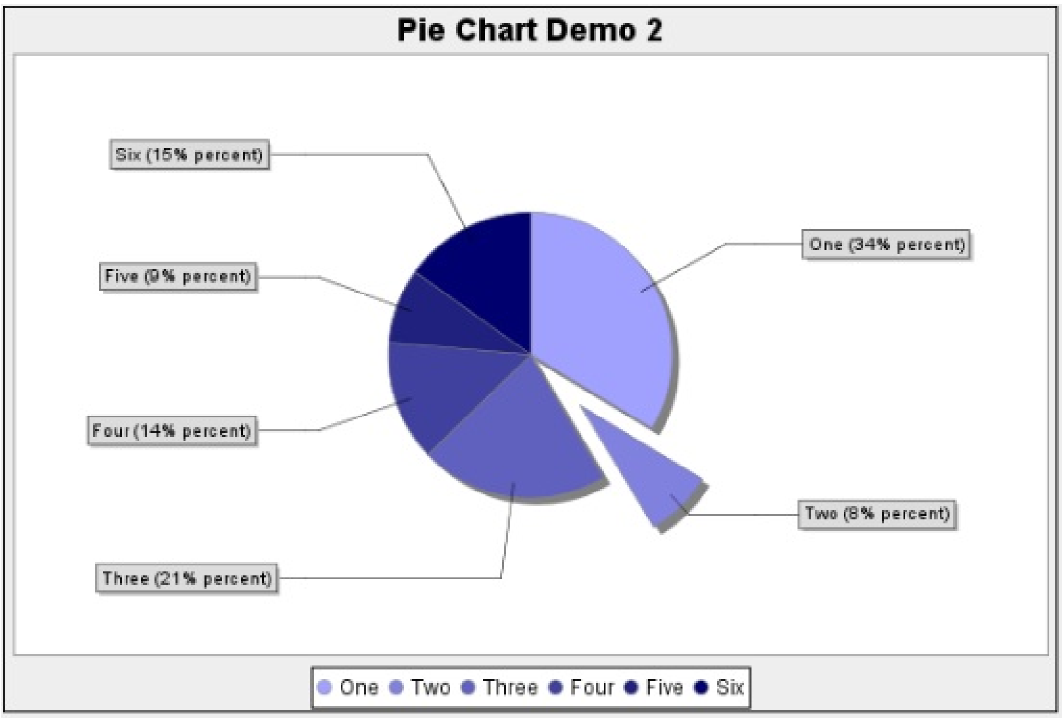


图2.2 一个带有取出“区域”的饼图（参见：PieChartDemo2.java）

我们也可以显示3D效果的饼图，如下图2.3所示：

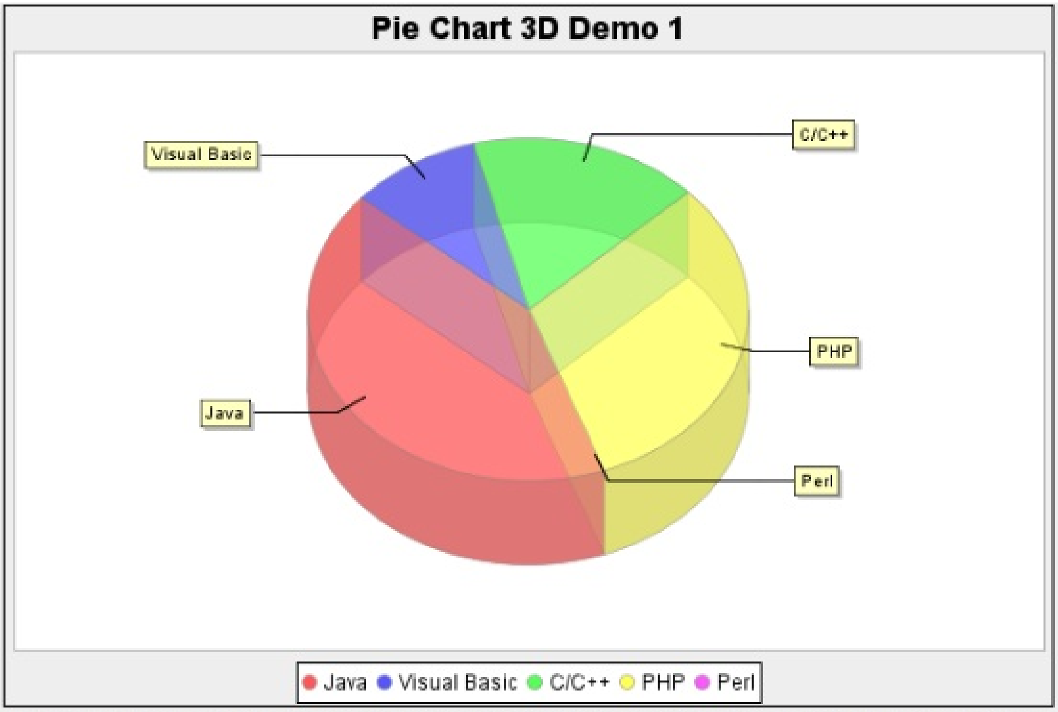


图2.3 3D效果图的图表（参见：PieChart3DDemo1.java）

3D效果的饼图，部分区域不能取出。

2.3 直方条形图（Bar Charts）

JFreeChart可以创建一系列的直方条形图。创建直方条形图的数据必须符合CategoryDataset接口标准。图2.4显示了一个垂直定向的直方条形图。

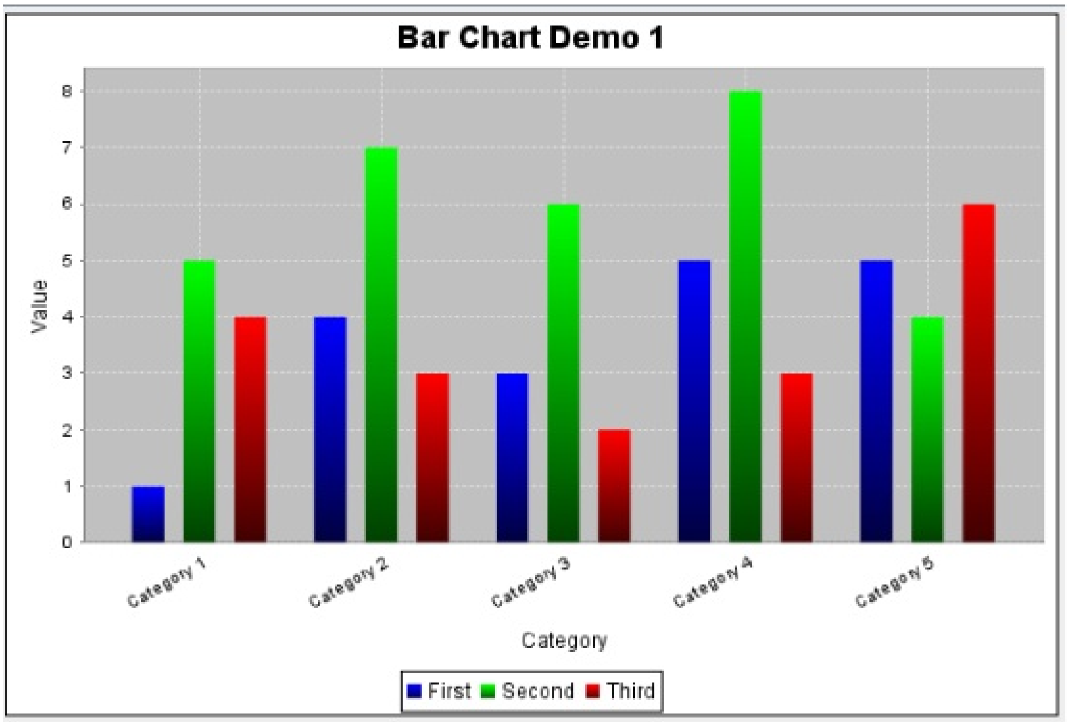
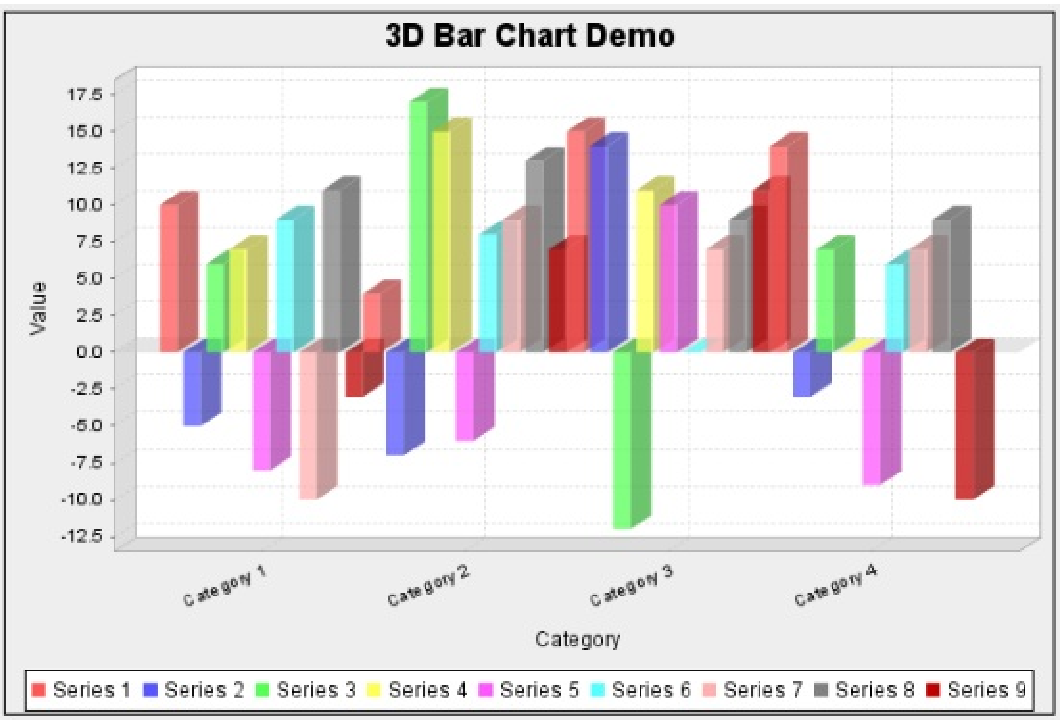


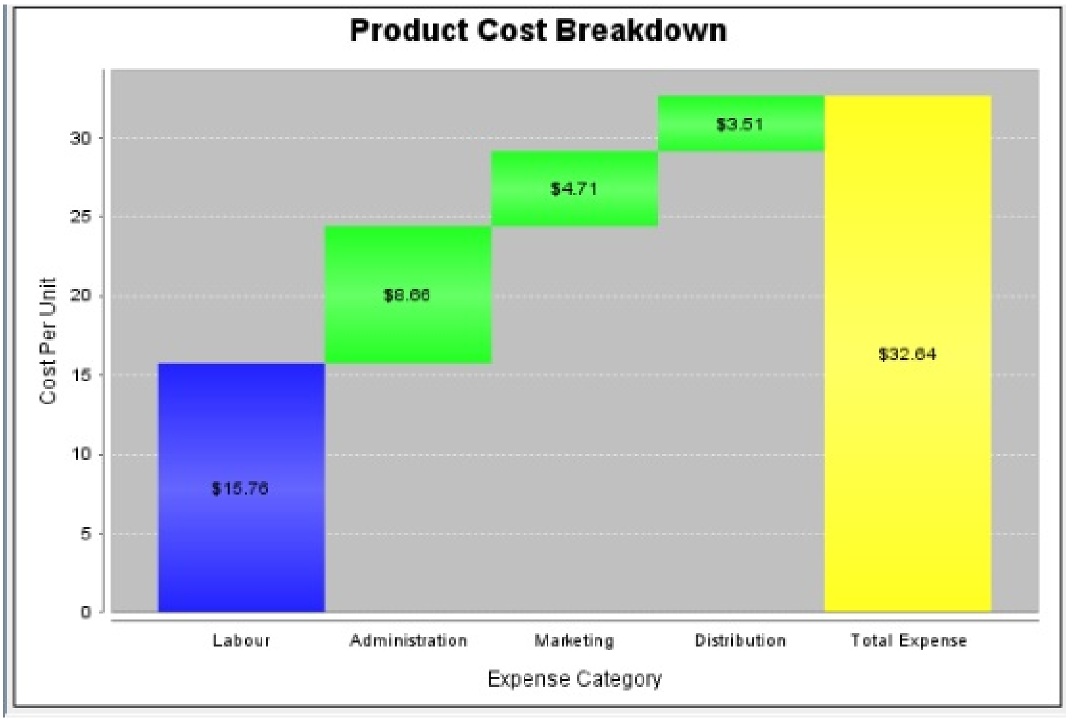
图2.4 一个垂直的直方条形图（参见：BarChartDemo1.java）

直方条形图可以用3D效果显示，如下图2.5所示。



如2.5 3D效果的直方条形图（参见：BarChart3DDemo1.java）

直方条形图的另一种变型，瀑布图表。如下图2.6所示：



如2.6 一个瀑布图表（参见：WaterfallChartDemo1.java）

直方条形图可以从时序数据中产生。如下图2.7所示：

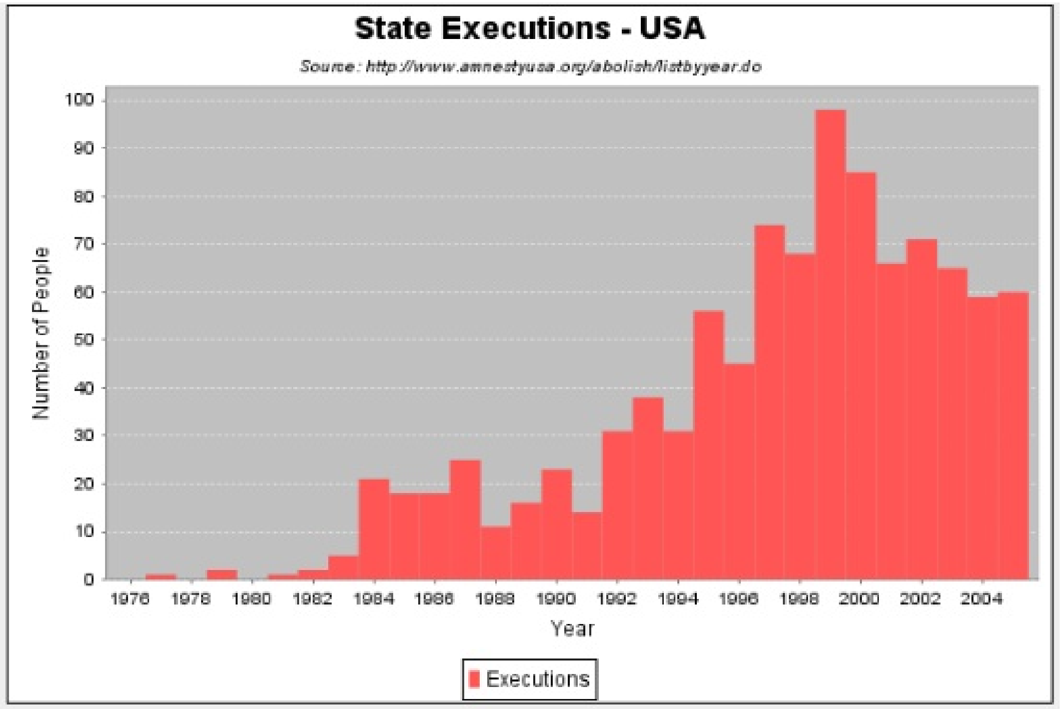


图2.7 一个XY图表（参考 XYBarChartDemo1.java）

2.4 折线图（Line Charts）

折线图可以使用直方条形图的数据对象CategoryDataset产生。如下图2.8：

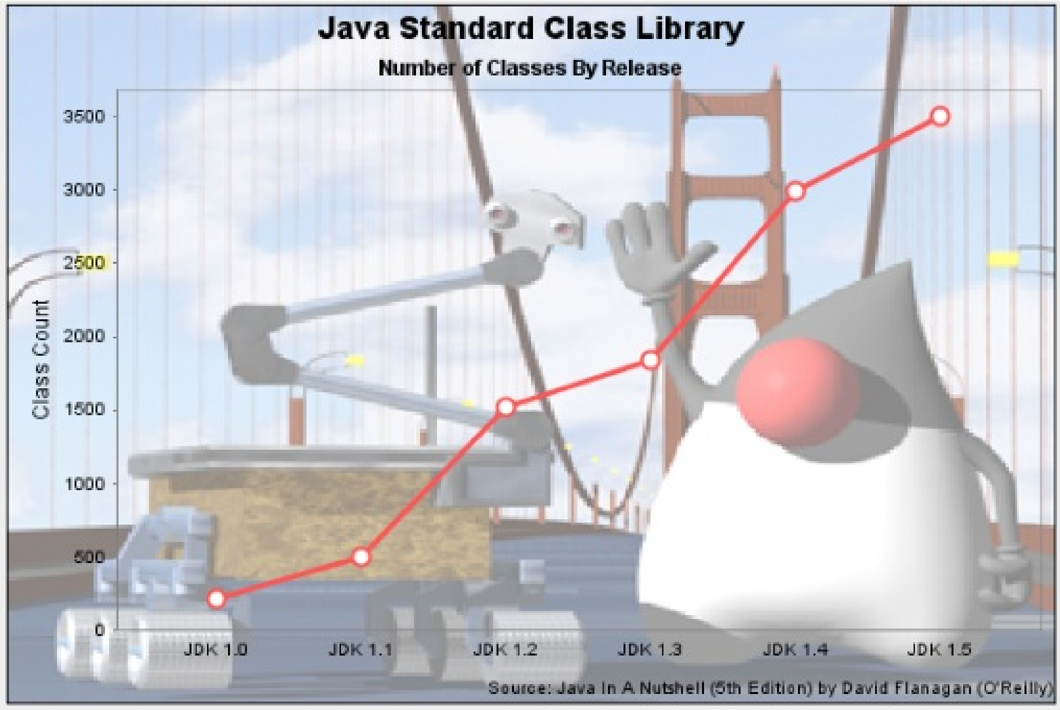


图2.8 一个折线图（参考LineChartDemo1.java）

2.5 XY(散点图)

XYDataset是第三种数据类型，用来产生一系列图表的类型。标准的XY区域有X和Y数轴。默认的，使用相应的数据按照一定比例画出X轴和Y轴。如图2.9所示。

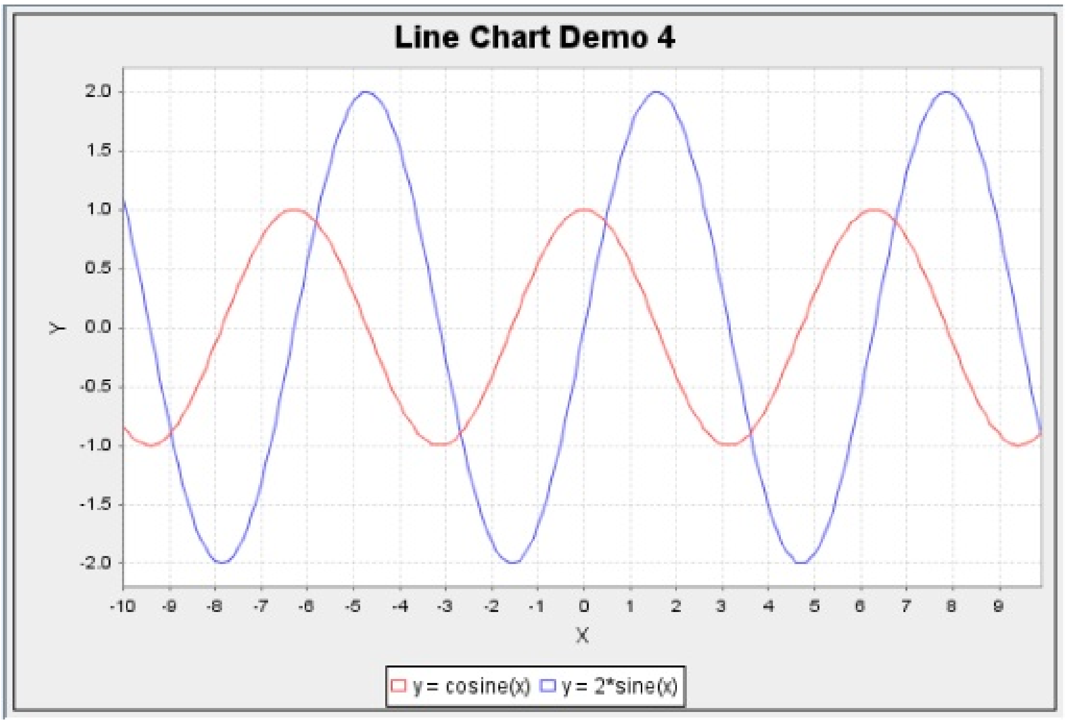


图2.9 折线图（参考：LineChartDemo4.java）

散点图是每一个数据点用一个图形画出来，而不是使用线将点连起来。一个实例如下图2.10所示：

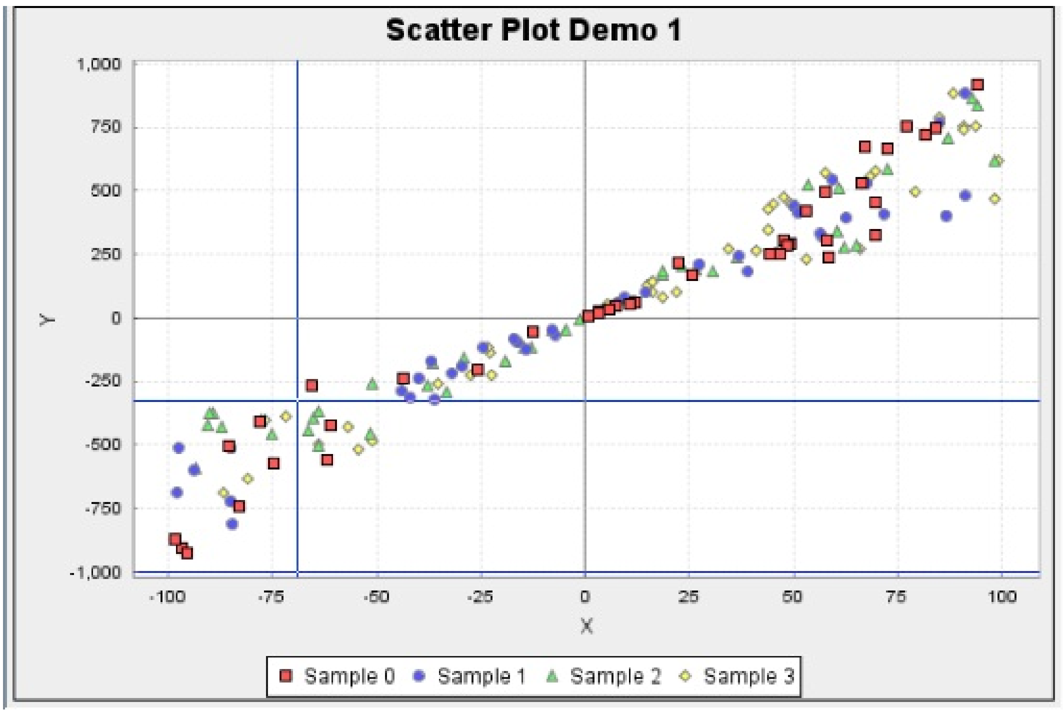


图2.10 散点图（参考：ScatterPlotDemo1.java）

2.6 时序图

JFreeChart支持时间序列图表，时序图包括平均值图、high-low-open-close图和candlestick图，如下图2.11所示：

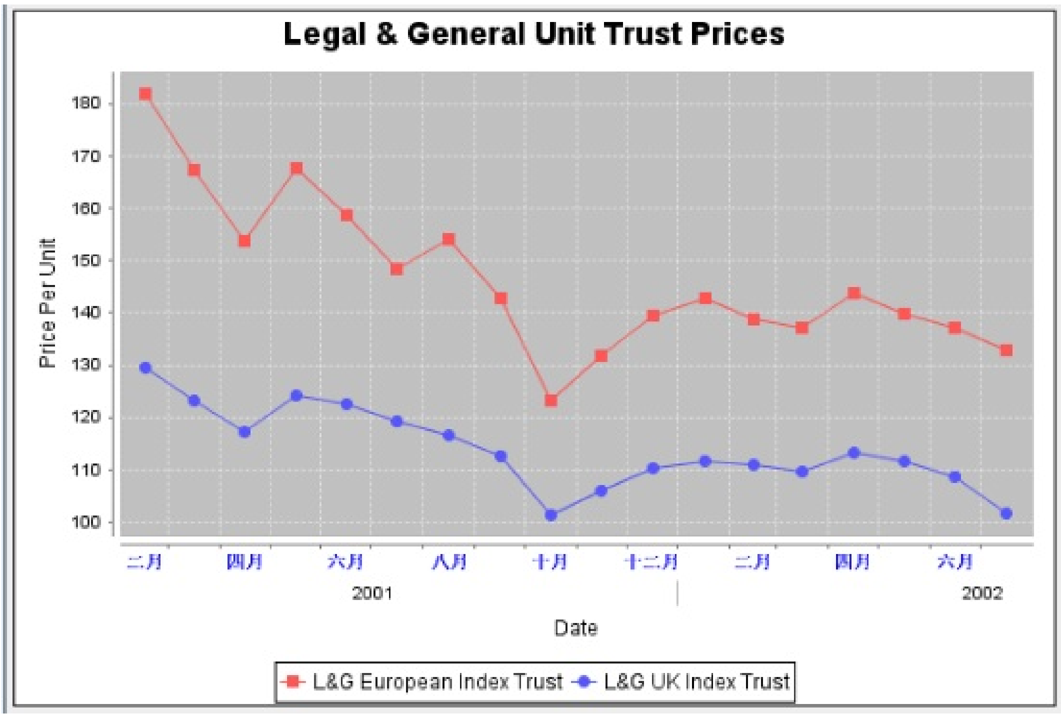


图2.11 序列图（参考：TimeSeriesDemo1.java）

我们可以在时序图上添加一条平均值线——如下图2.12所示：

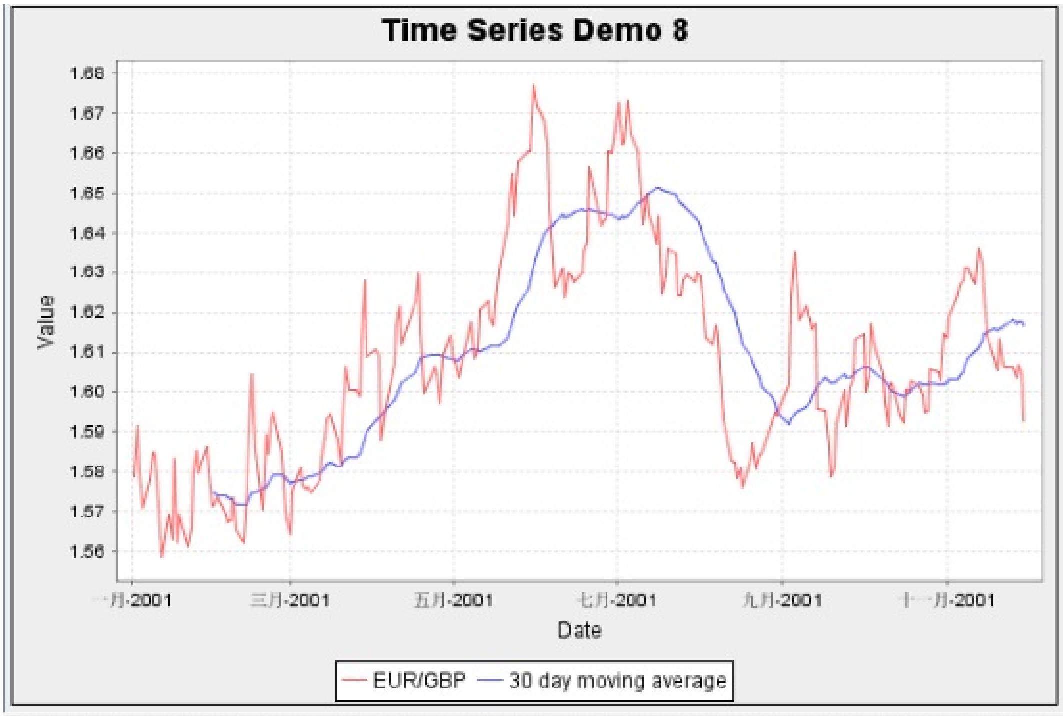


图2.12 带有平均线线的时序图（参考：TimeSeriesDemo8.java）

我们可以使用OHLCDataset（XYDataset的扩展）显示high-low-open-close数据图表。如下图2.13所示：

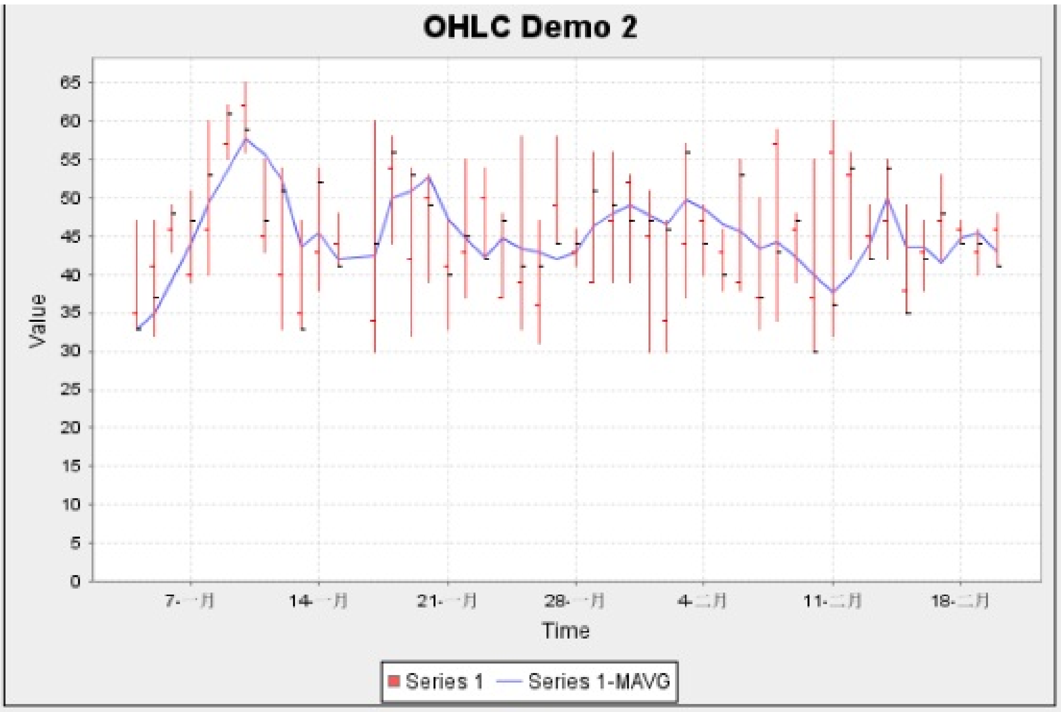


图2.13 high-low-open-close图表（参考：HighLowChartDemo2.java)）

2.7 柱状图

可以使用一个IntervalXYDataset（XYDataset的另一个扩展）数据产生柱状图。如下图2.14所示：

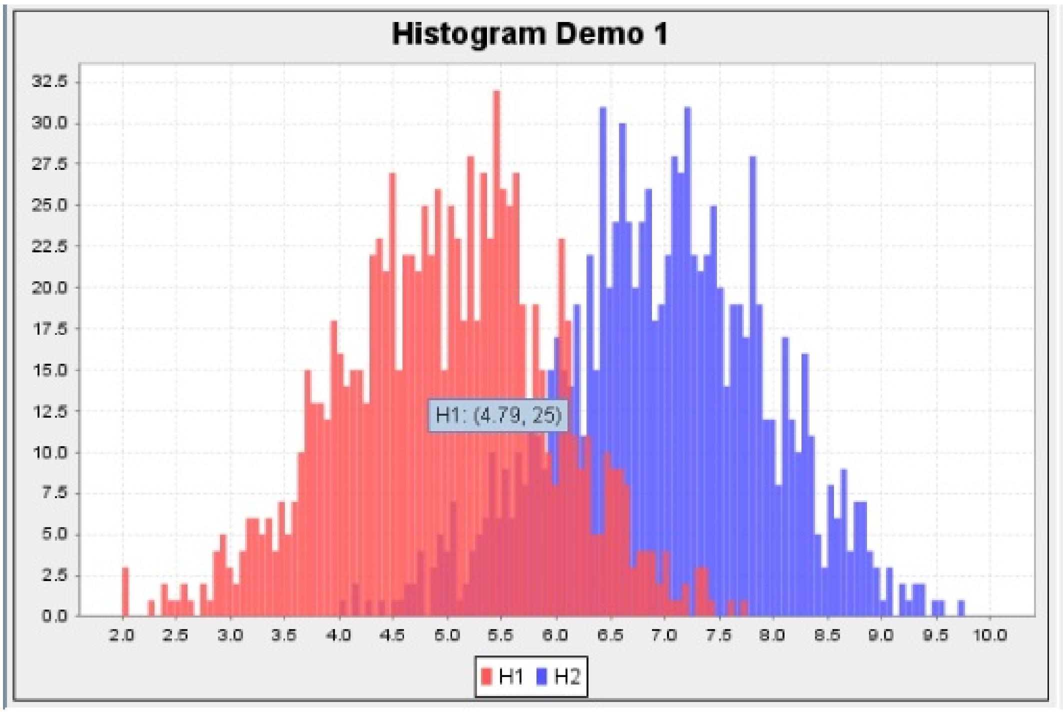


图2.14 柱状图（参考：HistogramDemo1.java）

2.8 面积图

我们可以使用CategoryDataset或者XYDataset产成面积图表。如下图2.15所示：

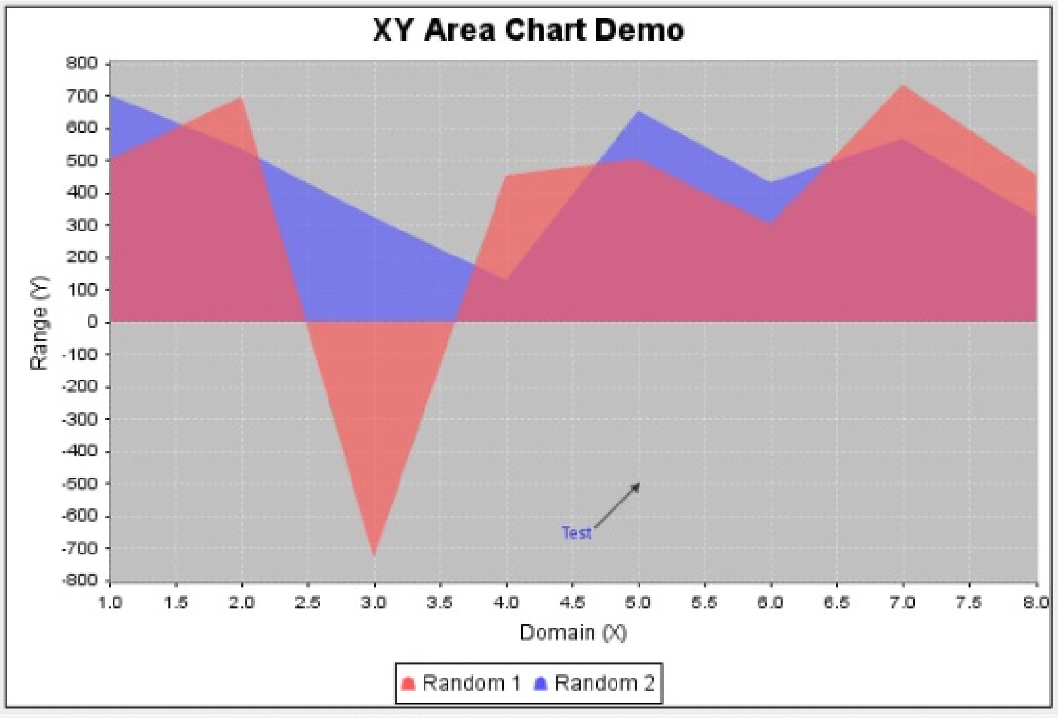


图2.15 面积图（参考：XYAreaChartDemo1.java）

同时，JFreeChart也支持堆栈式面积图表，如下图2.16所示：

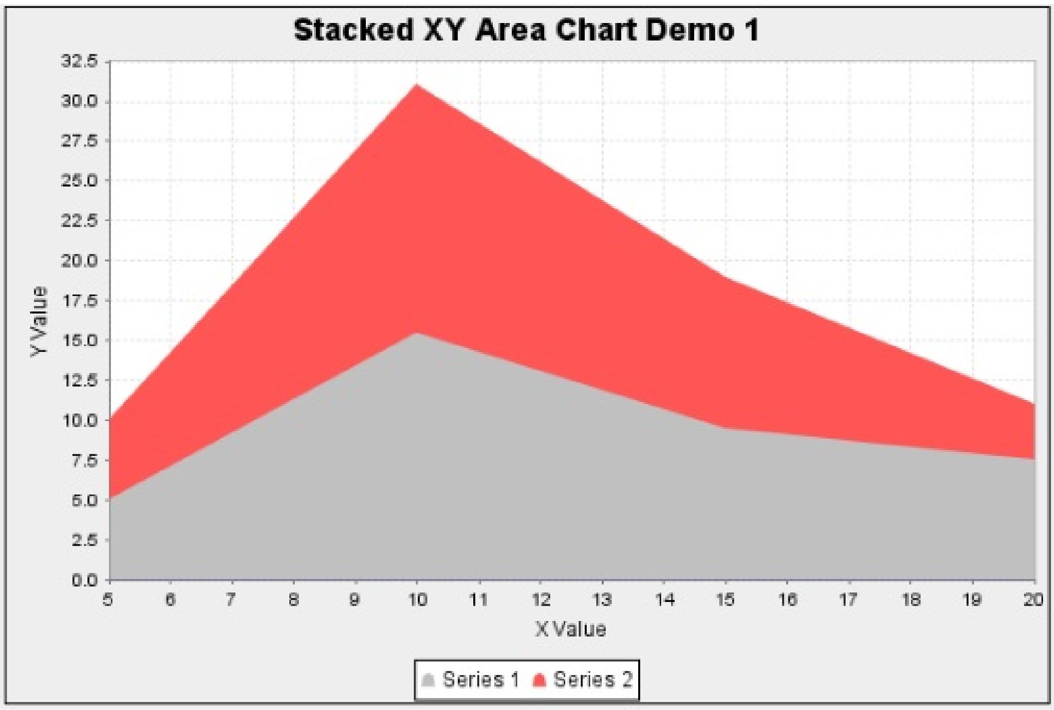


图2.16 堆栈式面积图（参考：StackedXYAreaChartDemo1.java）

2.9 差异图

差异图是显示两个序列之间的不同。如下图2.17所示。

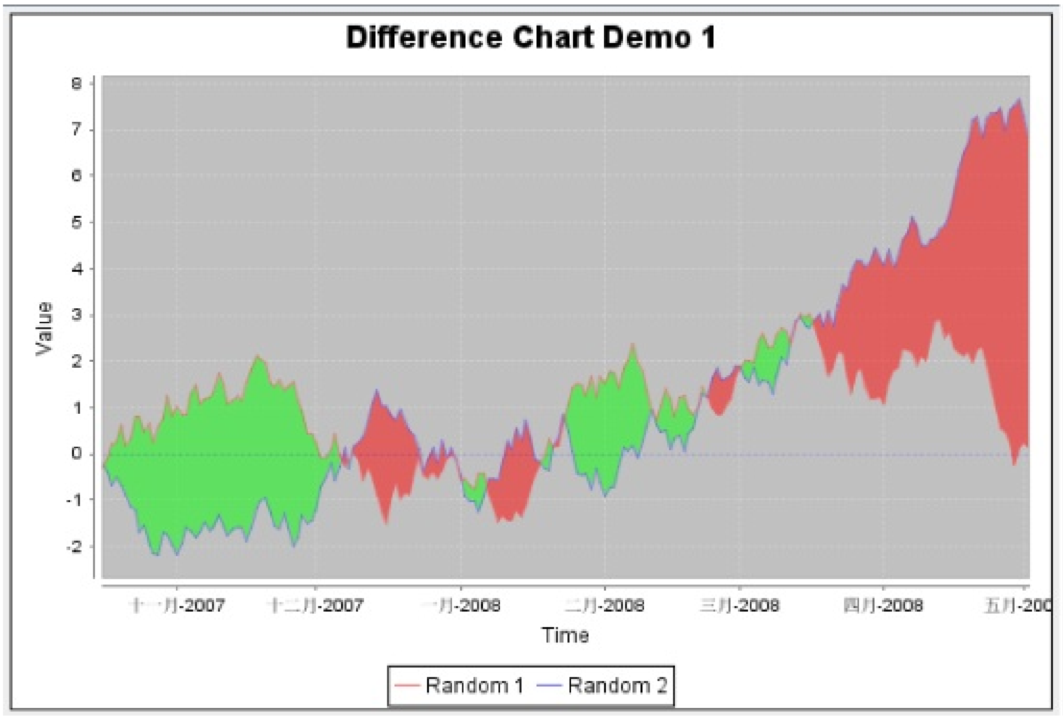


图2.17 差异图（参考：DifferenceChartDemo1.java）

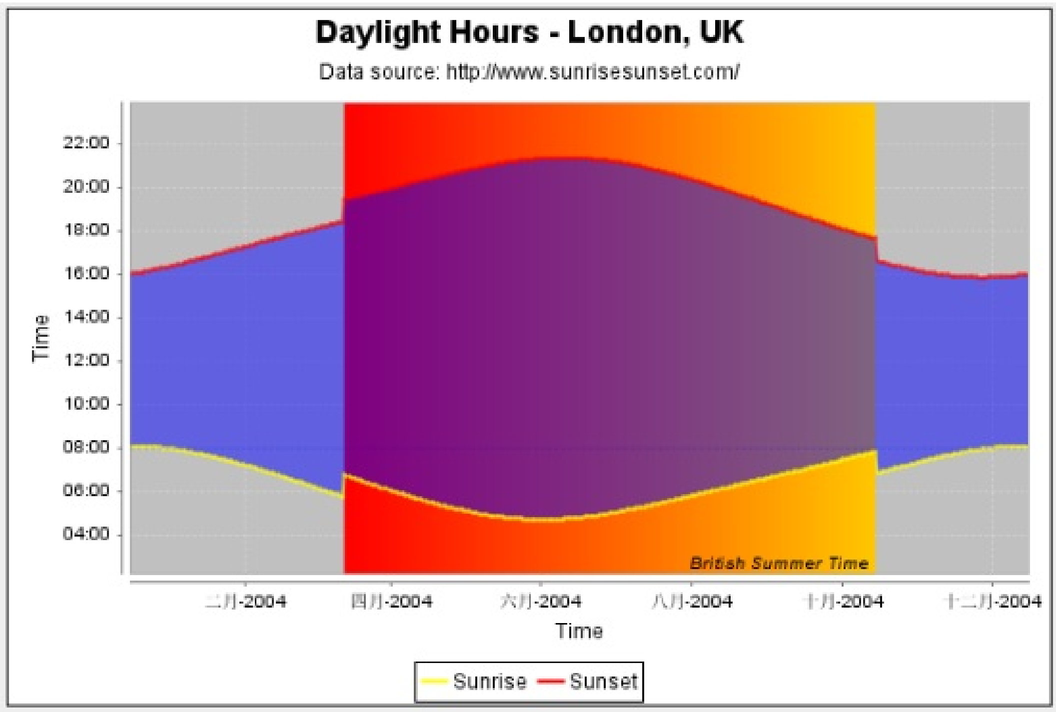


图2.18 差异图（参考：DifferenceChartDemo2.java）

2.10 梯形图

梯形图使用一系列的“梯形”来显示数据数值。——如下图2.19所示：

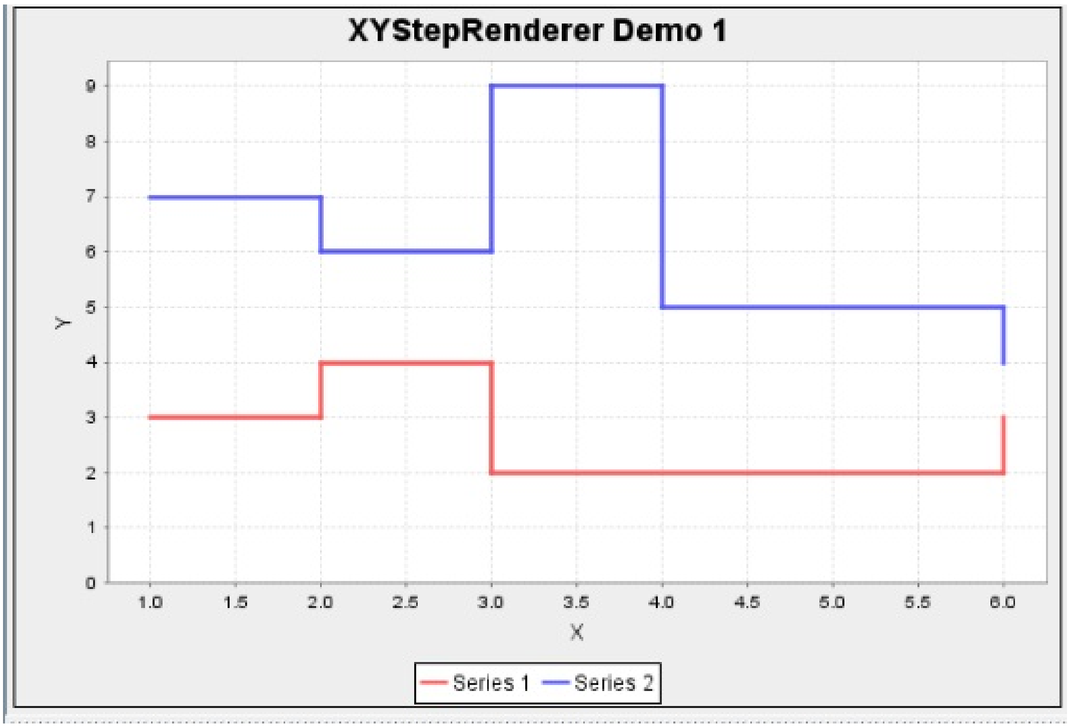


图2.19 梯形图（参考：XYStepRendererDemo1.java）

梯形图数据使用XYDataset数据对象。

2.11 甘特图

我们可以使用IntervalCategoryDataset数据集类产生甘特图。如图2.20所示

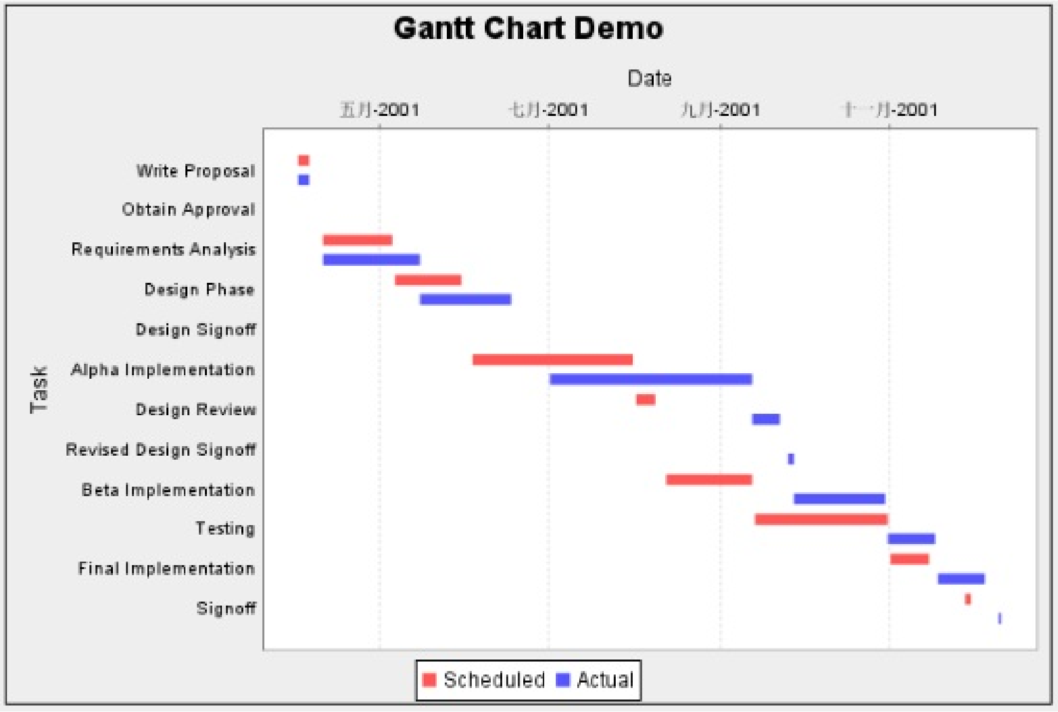


图2.20 甘特图（参考：GanttChartDemo1.java）

此外，甘特图可以具有子任务和进度显示器。如下图2.21所示

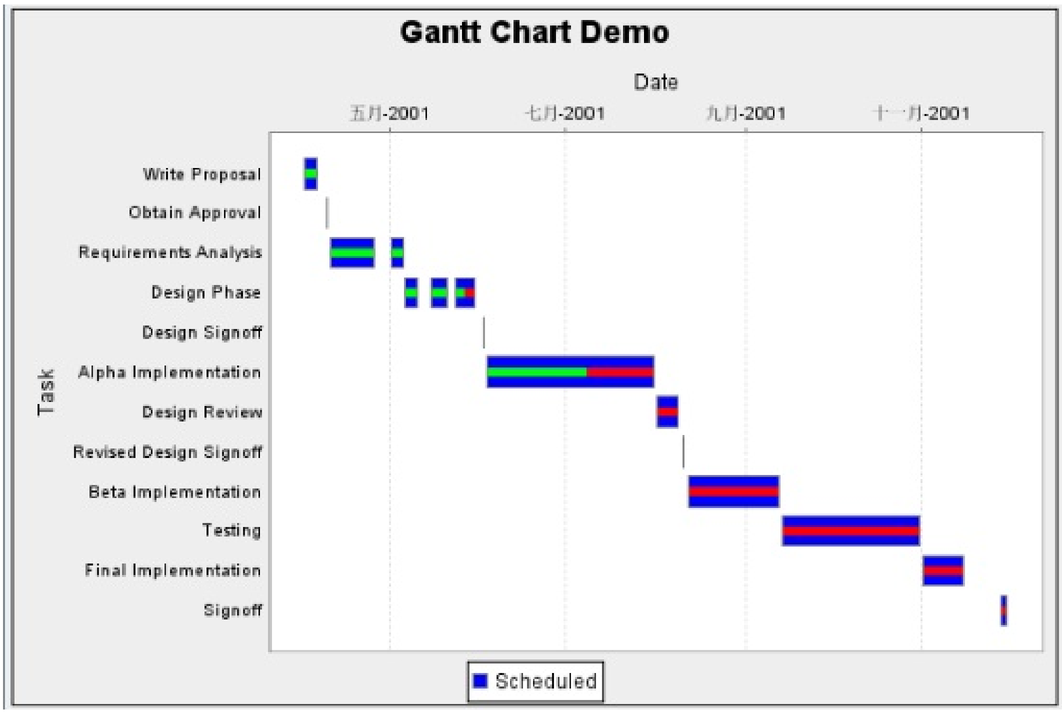


图2.21 带有进度显示的甘特图

2.12 多轴图

JFreeChart支持多轴图表。如下图2.22显示了一个价格—数量的图表。

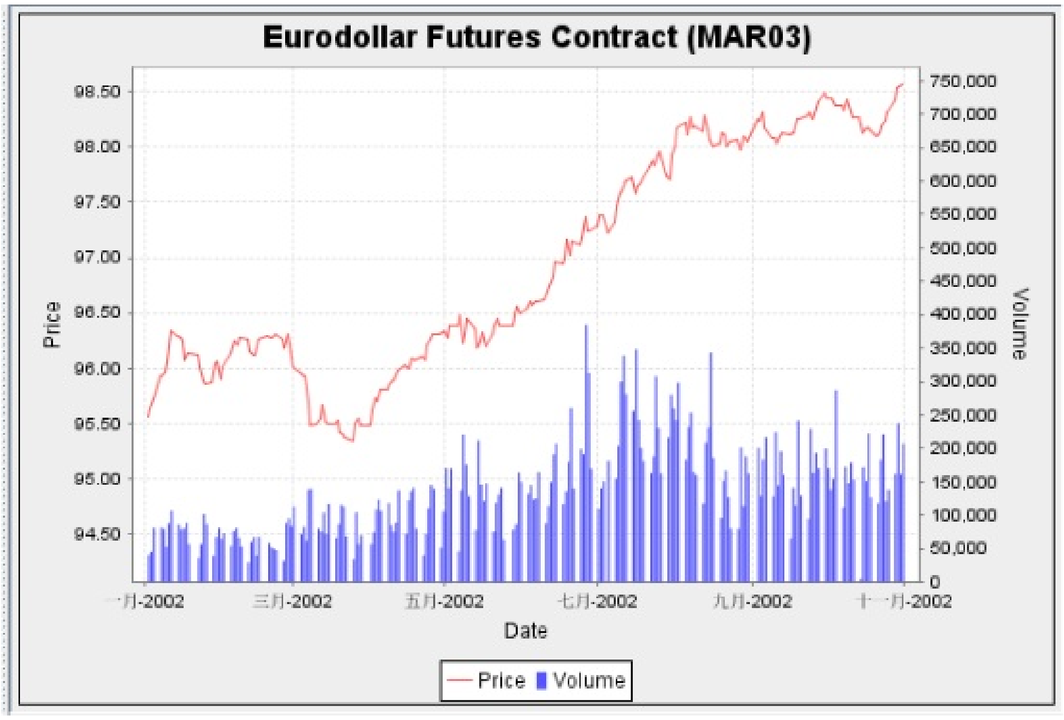


图2.22 价格—数量图表（参考：PriceVolumeDemo1.java）

CategoryPlot和XYPot支持多轴特征。图2.23显示了一个具有四个数轴的图表。

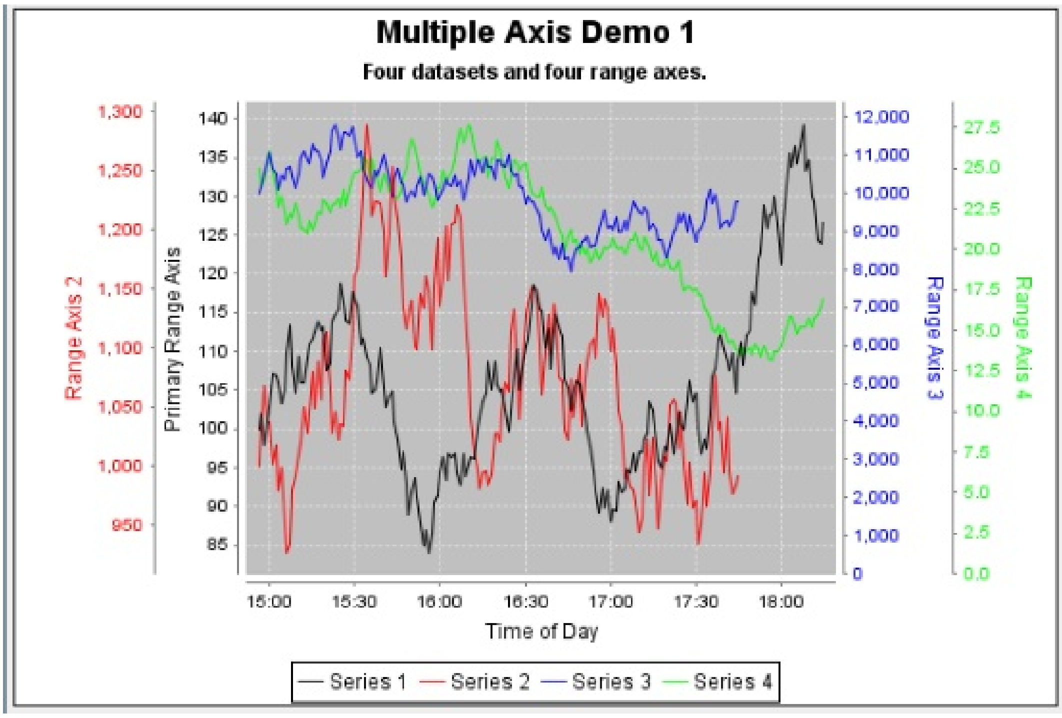


图2.23 多轴图表（参考：MultipleAxisDemo1.java）

2.13 复合/覆盖图

JFreeChart支持复合/覆盖图表。图2.24显示了一个条形图上覆盖了一个折线图。

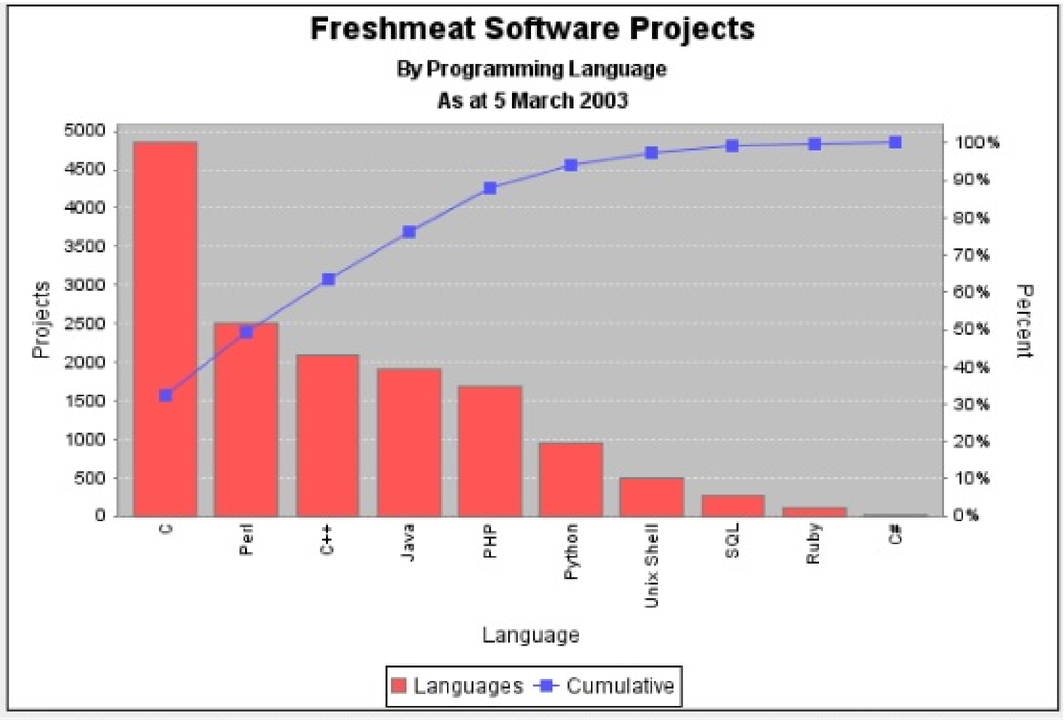


图2.24 覆盖图（参考：ParetoChartDemo1.java）

也有可能使用同一个主轴，组合几种图表。如下图2.25。

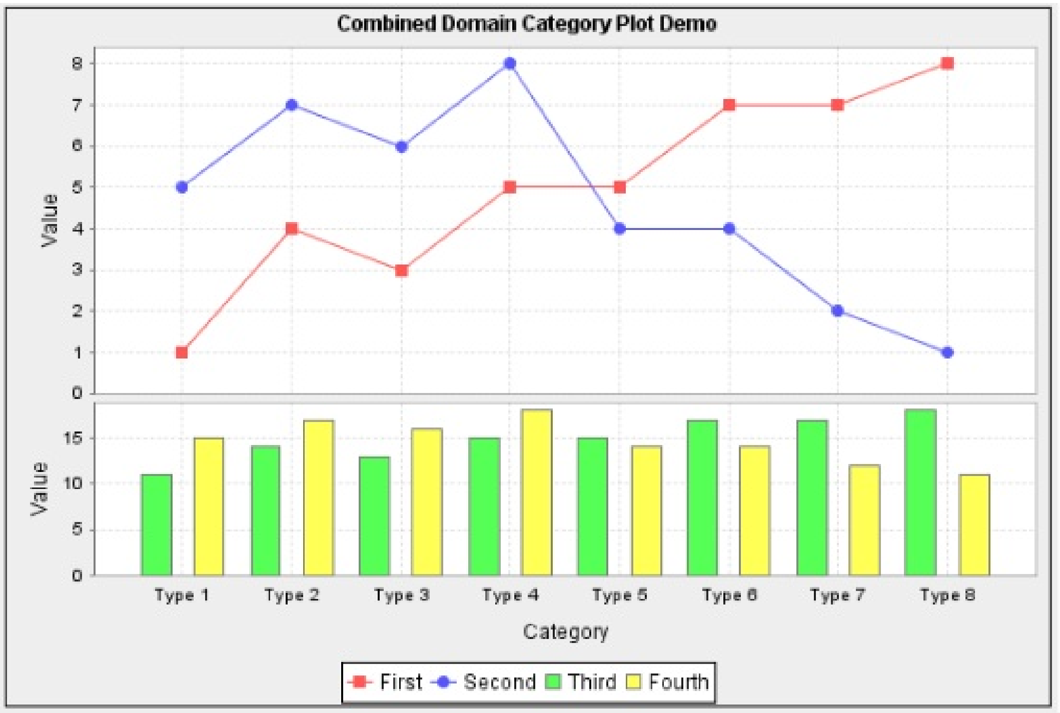


图2.25 带有公共区域的图表（参考：CombinedCategoryPlotDemo1.java）

类似的，JFreeChart可以复合几种图表，共用相同刻度范围的轴。如图2.26所示：

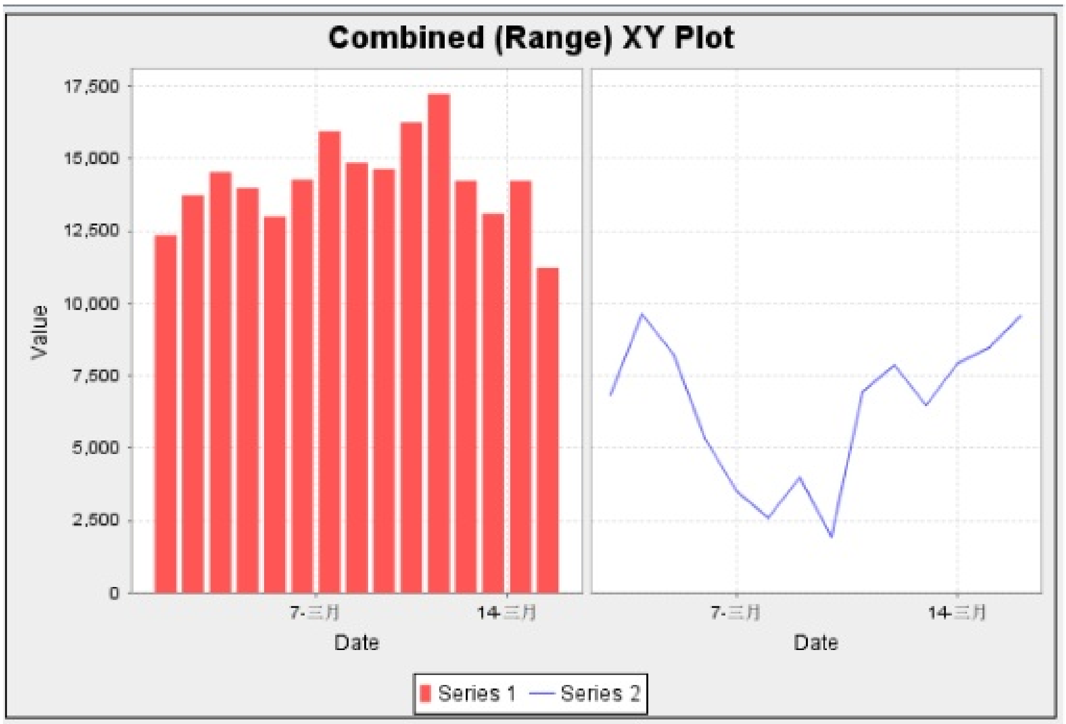


图2.26 共用相同范围轴的复合图表（参考：CombinedXYPlotDemo2.java）

2.14 开发远景

JFreeChart为免费软件，任何人可以扩展它，可以添加新的特征。已经有80多人向JFreeChart项目贡献代码。不久开发者将开发更多的图表来满足更多的需求。我们可以从JFreeChart网站上获得更多的信息和版本的更新：

http://www.jfree.org/jfreechart/

欢迎您的加入。

3 下载和安装JFreeChart 1.0.6

3.1 简介

本章主要介绍JFreeChart的下载、解包和编译内容。同时讲述如何运行JfreeChart应用实例，如何从源代码生成JavaDoc的HTML文件内容等。

3.2 下载

我们可以从JFreeChart主网站上获得最新的JFreeChart版本：

http://www.jfree.org/jfreechart/download

网页上提供了两个可下载的版本：

文件	描述
jfreechart-1.0.6.tar.gz	Linux/Unix版本
jfreechart-1.0.6.zip	Windows版本

这两个文件包含相同的源代码。主要不同在于下载的zip文件中所有的文本文件已被重新编码，有回车返回，并且每行结尾有换行符号。

JFreeChart是有JCommon（目前版本是1.0.9）基础类库.JCommon是运行时的jar文件，在JFreeChart下载时，包含该jar文件。如果你需要JCommon的源代码，可以从下面链接下载：

http://www.jfree.org/jcommon/

3.3 解包

下载完JFreeChart压缩文件之后，我们需要将该压缩文件解开。我们可以将文件解压到指定的目录下面。

3.3.1 Linux/Unix环境下解压

在Linux/Unix环境下解压文件，使用下面命令：

```
tar xvzf jfreechart-1.0.6.tar.gz tar xvzf jfreechart-1.0.6.tar.g
z
```

该命令将JFreeChart所有的文件包括源代码，运行jar文件和doc文档解压到jfreechart-1.0.6目录下面。

3.3.2 Windows环境下解压

在Windows环境下解压文件，使用下面命令：

```
jar -xvf jfreechart-1.0.6.zip
```

该命令将JFreeChart所有的文件包括源代码，运行jar文件和doc文档解压到jfreechart-1.0.6目录下面。

3.3.3 文件目录说明

解压后的jfreechart-1.0.6目录下面，有许多文件和文件夹，列表如下：

文件/目录	说明
Ant	该目录下面包含了一个ant的build.xml脚本。我们使用该脚本可以用现有的版本源代码重新构建JFreeChart。
checkstyle	该目录下面包含了几种检查风格属性文件。文件定义了JFreeChart中的编码规范。
experimental	该文件夹下面包含了一些不属于JFreeChart标准API的类文件。注意这些代码的API可能会改变。
gjdoc	该文件夹下面包含了一种产生JFreeChart文档的脚本。
lib	该目录下面包含了JFreeChart的jar文件，以及JFreeChart依赖的jar文件。
source	JFreeChart源代码目录。
swt	该目录下面包含了具有实践经验的swt源代码。注意该代码API有可能发生改变。
Tests	JFreeChart单元测试的源代码文件。
jfreechart-1.0.6-demo.jar	一个具有实例演示的可运行的jar文件。
CHANGELOG.txt	老的JFreeChart变更的日志记录。
ChangeLog	JFreeChart变更的详细日志记录。
licence-LGPL.txt	JFreeChart公共认证（GNU LGPL）
NEWS	JFreeChart项目新闻
README.txt	重要信息——一定要读！

我们应当花一部分时间来熟悉这些文件，并详细的阅读README.txt文件。

3.4 运行演示实例

在解压的文件中，有一个实例演示应用包含了JFreeChart产生的大量图表演示实例。输入下面命令可以运行该应用：

```
java -jar jfreechart-1.0.6-demo.jar
```

实例的源代码跟JFreeChart开发指南一并发行，是收费的。

3.5 编译源代码

我们可是使用ant的build.xml文件重新编译JFreeChart类文件。进入ant目录，然后键入：

```
ant complie
```

这将重新编译全部的源文件和创建创建JFreeChart运行时依赖的jar文件。Ant工具需要1.5.1版本或更高版本，我们可以从下面链接获得更多ant信息：

http://ant.apache.org/

3.6 产生javadoc文档

JFreeChart源代码文件中包含了大量丰富的Javadoc注释。我们使用javadoc工具可以直接从源代码中产生HTML文件。

产生javadoc文件使用ant的javadoc目标，进入ant目录键入：

```
ant javadoc
```

这将产生javadoc目录。目录下面包含了全部的Javadoc的HTML文件。

4 使用JFreeChart1.0.6

4.1 概述

本章节向使用JFreeChart的新用户编写了一个简单的实例，以说明JFreeChart的使用方法。

4.2 创建第一个图表

4.2.1 概述

使用JFreeChart创建图表共有三个步骤。如下：

- 创建一个dataset。该dataset包含图表要显示的数据。
- 创建一个JFreeChart对象。该对象负责画这个图表。
- 创建一个输出目标（如：一个panel，显示在屏幕上）。该输出目标画这个图表。

下面，我们使用一个简单的应用（First.java）来描述这个过程。该应用产生了一个饼图，如下图4.1所示：

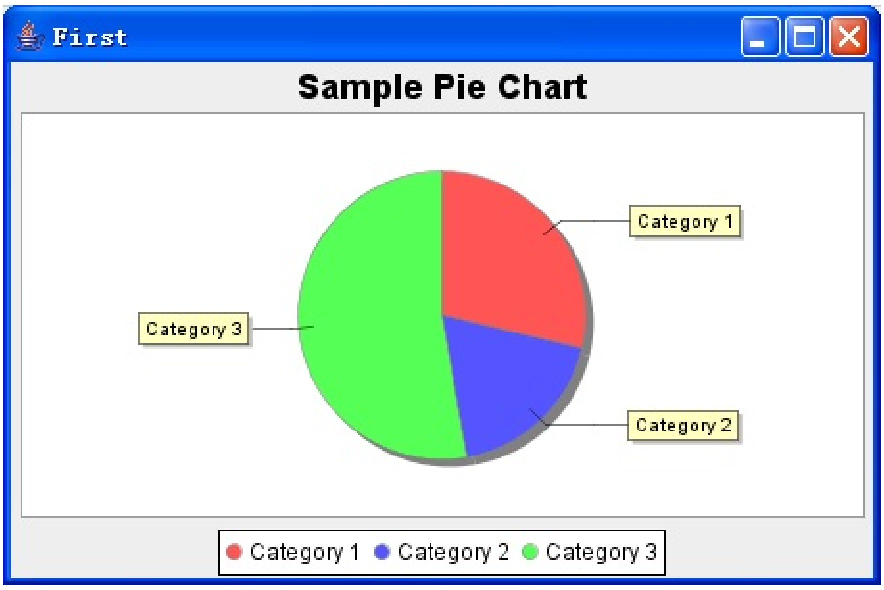


图4.1 创建的第一个饼图（参考First.java）

上面描述的三个步骤，将在下面的章节里面，均有代码详细说明。

4.2.2 数据

步骤一要求我们为我们的图表创建一个dataset。使用DefaultPieDataset类可以很容易创建。如下代码：

```
// create a dataset...
DefaultPieDataset dataset = new DefaultPieDataset();
dataset.setValue("Category 1", 43.2);
dataset.setValue("Category 2", 27.9);
dataset.setValue("Category 3", 79.5);
```

注意：

JFreeChart可以使用符合PieDataset接口的任何实现数据来创建饼图。DefaultDataset类实现了PieDataset接口，提供了一种便利的使用方式。

我们可以自由的开发符合实际需的任意PieDataset接口实现。

4.2.3 创建一个饼图

步骤二关心的是我们如何使用这个dataset展示在区域中。这就需要我们创建一个JFreeChart对象，该对象使用我们的饼图dataset数据画一个图表。我们使用ChartFactory类来创建，代码如下：

```
// create a chart...
JFreeChart chart = ChartFactory.createPieChart(
    "Sample Pie Chart",
    dataset,
    true, // legend?
    true, // tooltips?
    false // URLs?
);
```

注意：

代码中将一个dataset的引用传入到工厂方法中。JFreeChart持有这个dataset引用的目的是便于在画图表时能够获得数据。使用JFreeChart创建图表有许多定制外观的方式，在这个例子中我们使用缺省的属性值。后面章节将详细介绍。

4.2.4 显示图表

最好一个步骤就是在某个地方显示该图表。JFreeChart提供了非常灵活的图表输出方式。

现在我们可以在一个屏幕的框架中显示这个图表。ChartFrame具有显示图表的机制（ChartPanel）。代码如下：

```
// create and display a frame...
ChartFrame frame = new ChartFrame("First", chart);
frame.pack();
frame.setVisible(true);
```

代码全部完成，运行main()方法,可以出现图4.1界面。

4.2.5 全部程序代码

下面是整个例子的全部代码，更加清楚的看到我们需要导入的类包和实现方法。

```
public class First {
    public static void main(String[] args) {
        // create a dataset...
        DefaultPieDataset dataset = new DefaultPieDataset();
        dataset.setValue("Category 1", 43.2);
        dataset.setValue("Category 2", 27.9);
        dataset.setValue("Category 3", 79.5);
        // create a chart...
        JFreeChart chart = ChartFactory.createPieChart(
            "Sample Pie Chart",
            dataset,
            true, // legend?
            true, // tooltips?
            false // URLs?
        );
        // create and display a frame...
        ChartFrame frame = new ChartFrame("First", chart);
        frame.pack();
        frame.setVisible(true);
    }
}
```

5 饼图（Pie Charts）

5.1 简介

本章主要讲解JFreeChart中饼图的一些特征。内容如下：

- 控制颜色和饼图片区的外廓
- null值和零值的处理
- 饼图片区的标签（定制文本，改变分配的比例空间）
- “取出”某个片区
- 多个饼图显示
- 显示3D效果的饼图

更多的信息，可以参见PiePlot参考文档，见章节33.72。

5.2 创建一个简单的饼图（Pie Charts）

在前面的第四章的手把手的向导里创建了一个简单的饼图。在这里不做详细介绍。

5.3 片区颜色

饼图片区缺省填充的颜色是自动分配的，正如你上面实例看到的。如果你不喜欢这个缺省的颜色，你可以实用setSectionPaint（）方法来设置片区颜色。例如：

```
PiePlot plot = (PiePlot) chart.getPlot();
plot.setSectionPaint("Section A", new Color(200, 255, 255));
plot.setSectionPaint("Section B", new Color(200, 200, 255));
```

JFreeChart的实例PieChartDemo2.java演示了如何定制颜色。在JFreeChart的代码中，片区颜色使用三层色属性机制来定义的。同时，我们也可以对饼图中的每一个系列定义填充的颜色，这里我们不做细述，更多的信息请参阅PiePlot类（33.27章节）。

5.4 片区外廓

每一个饼图片区的外廓默认是一条细灰线勾画出来的。PiePlot类提供了如下选择项：

- 完全不显示片区外廓
- 通过改变缺省的值来改变全部的片区外廓
- 单独改变部分饼图的片区外廓

5.4.1 片区外廓的可见性控制

为了完全关闭片区外廓，使用下面代码：

```
PiePlot plot = (PiePlot) chart.getPlot();
plot.setSectionOutlinesVisible(false);
```

在任何时候，你只需要使用下面代码可以让外廓显示出来：

```
plot.setSectionOutlinesVisible(true);
```

调用该方法可以触发PlotChangeEvent事件。

5.4.2 片区外廓的控制

在片区外廓显示的时候，我们可以改变饼图片区的整个外廓颜色或风格或者单个饼图片区的颜色或风格。整个外廓颜色或风格的修改需要在基本层里面设置，单个饼图片区的颜色设置需要在系列层中设置。在基本层里，如果没有更高层的颜色设置，则调用已定义的默认设置。我们可以使用PiePot类的方法来改变我们的设置。如下方法：

```
public void setBaseSectionOutlinePaint(Paint paint);
public void setBaseSectionOutlineStroke(Stroke stroke);
```

有时候在图表里面，我们会更喜欢设置饼图里面某个具体的片区的外廓的颜色，或许突出显示某些片区的细节方面。做到这些，我们可以是使用系列层层设置，通过下面的方法来定义。

```
public void setSectionOutlinePaint(Comparable key, Paint paint);
public void setSectionOutlineStroke(Comparable key, Stroke strok
e);
```

方法的第一个参数是dataset的片区关键值。如果我们将该值设为null，则系统将使用基本层的设置。

5.5 空置、零值和负值

PieDataset可能会包含一些饼图不可能显示的数值，比如null、零值或者负值。对于这些数据PiePlot类有专门的处理机制来处理。如果是零值，并且该值有意义，PiePlot类默认将一个标签放置在饼图片区显示的位置，并且在图表的图例中添加一个分类。如果零值可以忽略，我们可以使用下面代码设置一个标志，不显示该数据：

```
PiePlot plot = (PiePlot) chart.getPlot();
plot.setIgnoreZeroValues(true);
```

类似的null值也是如此处理，nul值代表dataset丢失或者不知来源的值。缺省的处理与零值相同，如果忽略null值，则代码如下：

```
PiePlot plot = (PiePlot) chart.getPlot();
plot.setIgnoreNullValues(true);
```

在饼图中处理负值是非常不明知的 ，所以在JFreeChart中负值总是被忽略的。

5.6 片区和图例标签

片区标签使用的文本，即可以在图表上显示，也可以在图表的图例上显示，并且完全可以定制。标签是自动默认产生的，但我们可以使用下面方法来改变：

```
public void setLabelGenerator(PieSectionLabelGenerator generator
);
public void setLegendLabelGenerator(PieSectionLabelGenerator gen
erator);
```

StandPieSectionLabelGenerator类专门用来生成图例的一个实现类，提供灵活处理定制标签的功能（如果你不喜欢用这个类，可以定义自己的类，只要实现接口PieSectionLabelGenerator即可）。Dataset显示出的标签值由Javade信息格式类来进行格式化——表5.1所示格式化的变量值。

名称	描述
{0}	片区关键值（字符串）
{1}	片区值
{2}	百分比的片区值

表5.1 StandardPieSectionLabelGenerator substitutions

下面举例说，假如我们有一个PieData包含下面的值

片区标识	片区值
S1	3.0
S2	5.0
S3	Null
S4	2.0

表 5.2 一个dataset实例

下面是格式化字符串产生的标签值内容：

格式化字符串	片区	产生的标签值
{0}	0	S1
{0} has value {1}	1	S2 has value 5.0
{0}({2} percent)	0	S1(30 percent)
{0} = {1}	2	S3 = null

类PieChartDemo2.java使用了定制标签的方法。

5.7 “取出”某个片区

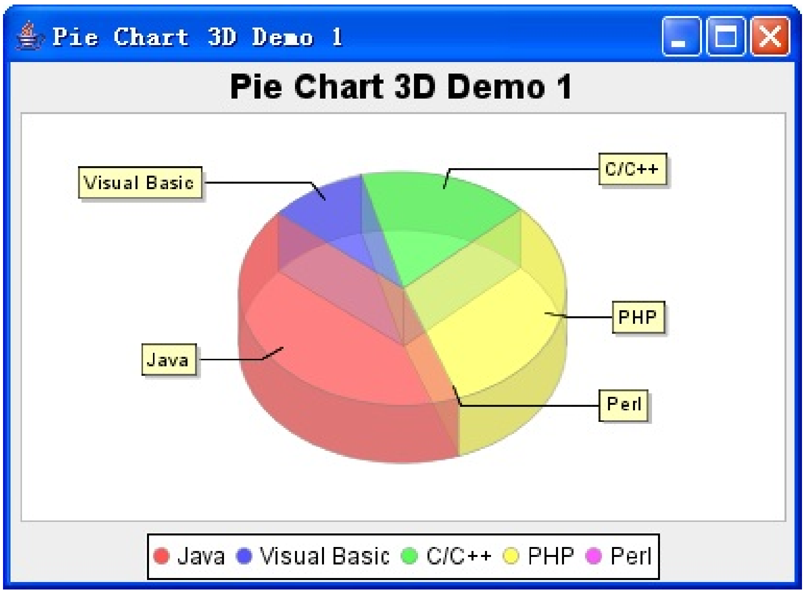
PiePlot类支持将某个片区“取出“显示。即某个片区偏离图表中心，以突出显示。如类所显示。

片区偏离的数值是图表半径的一个百分值来表示。例如0.3（30 persent）代码偏离的值是半径的长度×0.3.代码如下：

```
PiePlot pieplot = (PiePlot) jfreechart.getPlot();
pieplot.setExplodePercent("Two", 0.5);
```

5.8 3D饼图

JFreeChart具有一个实现3D效果的饼图类PiePlot3D，如图5.5所示，(PieChart3DDemo1.java)。PiePlot3D是PiePlot的子类，因此在我们创建自己的饼图时，使用PiePlot3D替换掉原来的PiePlot即可。创建3D效果的饼图时，使用ChartFactory的 createPieChart3D（）方法，而不是createPieChart（）方法。



如5.5 3D效果图。

对于该类有一些限制，如下：

- 不支持"取出"片区功能。
- 不支持轴项转动——如果支持，3D效果图可能会变型。

3D的实例主要是类PieChart3DDemo1-3.java。讲解类中没有列出其他两个。因为功能雷同于非3D效果。

5.9 多饼图

我们可是使用类MultiplePiePlot在一个图表上显示多个饼图。饼图的数据使用CatoryDataset。如图5.6所示。每个独立的饼图由一个专门的图表多次创建而成。创建的每一个饼图的PieDataset是由系统提供的CategoryDataset按照行或者列拆分出来的。代码见5.10.11.

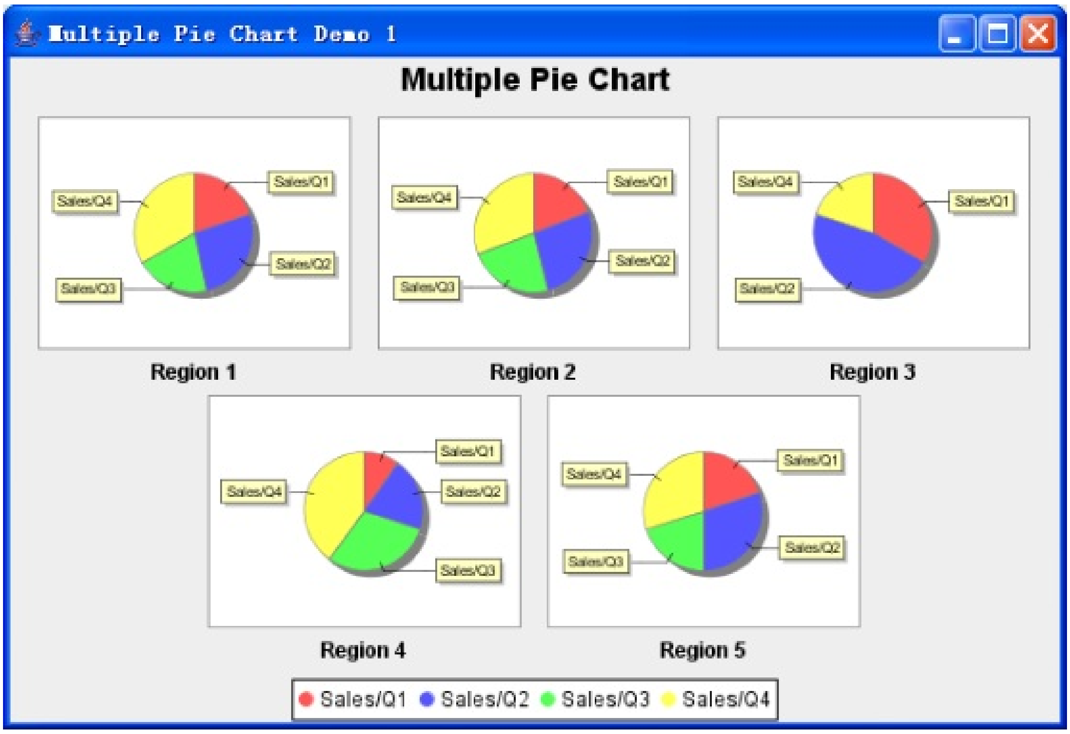


图5.6 多饼图图表

5.10 实例讲解

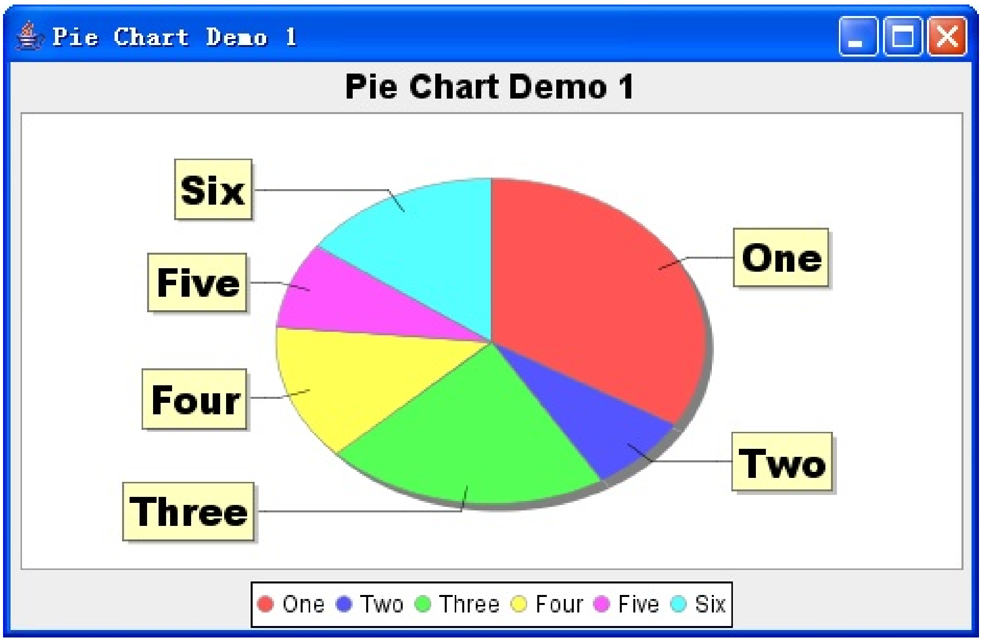
5.10.1 体会

也分为三层界面显示层、数据层、控制层。

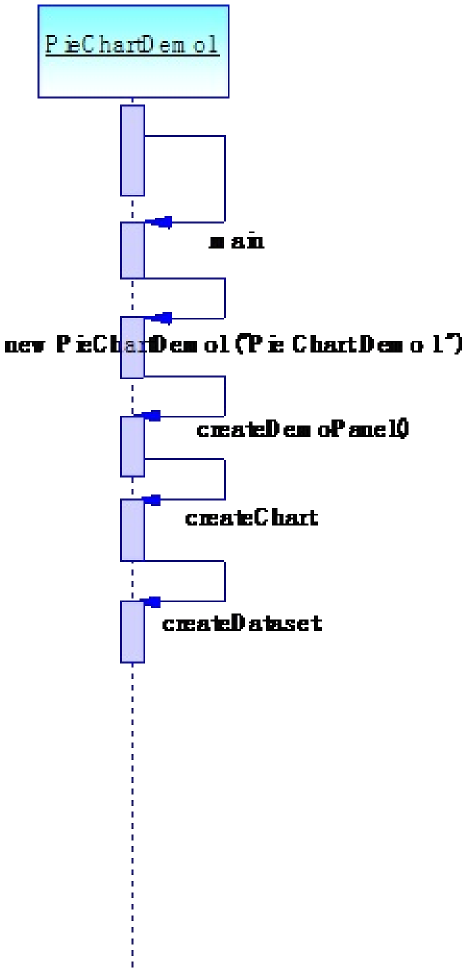
- 数据层比较复杂。每个图形有不同的数据类型。Dataset类控制。
- 控制层Plot，饼图使用PiePlot设置显示的图形的面貌的控制，JFreeChart最丰富的功能。
- 设置片区颜色
- 设置标签（文本格式、背景颜色等）
- 设置是否取出某块。
- 设置饼图是否为圆形
- 设置饼图旋转。
- 显示层JFreeChart使用数据、控制直接将数据显示出来。

5.10.2 类PieChartDemo1.java

效果图如下：



代码编写典型的使用了面对对象的方法：



全部代码如下：

```
package demo;
import java.awt.Dimension;
import java.awt.Font;
import javax.swing.JPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PiePlot;
import org.jfree.chart.title.TextTitle;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.general.PieDataset;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;
public class PieChartDemo1 extends ApplicationFrame {
    /**
    *
    */
    private static final long serialVersionUID = 259855755772408
5474L;
    public PieChartDemo1(String string) {
        super(string);
        JPanel jpanel = createDemoPanel();
        jpanel.setPreferredSize(new Dimension(500, 270));
```

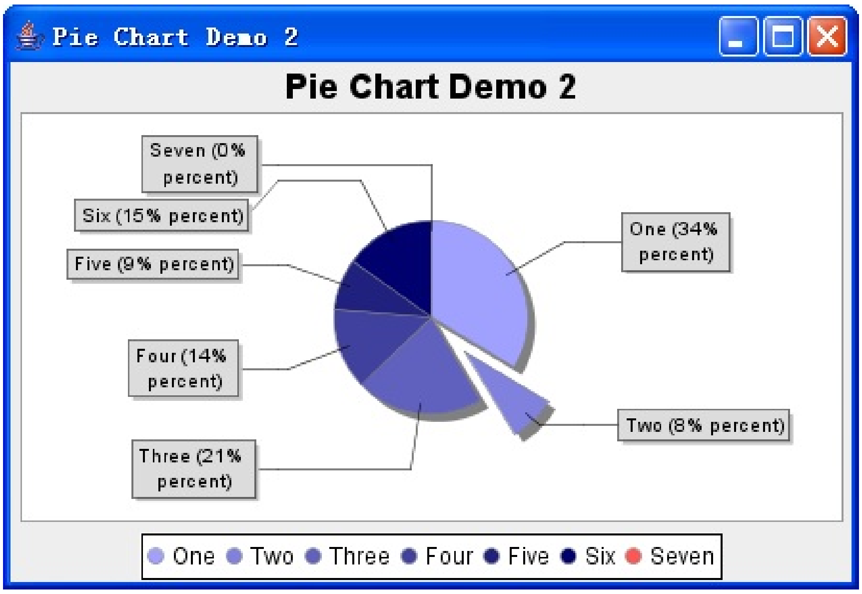
```
        setContentPane(jpanel);
    }
    private static PieDataset createDataset() {
        DefaultPieDataset defaultpiedataset = new DefaultPieData
set();
        defaultpiedataset.setValue("One", new Double(43.2));
        defaultpiedataset.setValue("Two", new Double(10.0));
        defaultpiedataset.setValue("Three", new Double(27.5));
        defaultpiedataset.setValue("Four", new Double(17.5));
        defaultpiedataset.setValue("Five", new Double(11.0));
        defaultpiedataset.setValue("Six", new Double(19.4));
        return defaultpiedataset;
    }
    private static JFreeChart createChart(PieDataset piedataset)
 {
        JFreeChart jfreechart = ChartFactory.createPieChart("Pie
 Chart Demo 1",
        piedataset, true, true, false);
        TextTitle texttitle = jfreechart.getTitle();
        texttitle.setToolTipText("A title tooltip!");
        PiePlot pieplot = (PiePlot) jfreechart.getPlot();
        pieplot.setLabelFont(new Font("Arial Black", 0, 20));
        pieplot.setNoDataMessage("No data available");
        pieplot.setCircular(false);
        pieplot.setLabelGap(0.02);
        return jfreechart;
    }
    public static JPanel createDemoPanel() {
        JFreeChart jfreechart = createChart(createDataset());
        return new ChartPanel(jfreechart);
    }
    public static void main(String[] strings) {
        PieChartDemo1 piechartdemo1 = new PieChartDemo1("Pie Cha
rt Demo 1");
        piechartdemo1.pack();
        RefineryUtilities.centerFrameOnScreen(piechartdemo1);
        piechartdemo1.setVisible(true);
    }
}
```

5.10.3 类PieChartDemo2.java

功能：

“取出“片区显示。

效果：



代码：

```
private static JFreeChart createChart(PieDataset piedataset) {
    JFreeChart jfreechart = ChartFactory.createPieChart("PieChar
t Demo 2", piedataset, true, true, false);
    PiePlot pieplot = (PiePlot) jfreechart.getPlot();
    pieplot.setSectionPaint("One", new Color(160, 160, 255));
    pieplot.setSectionPaint("Two", new Color(128, 128, 223));
    pieplot.setSectionPaint("Three", new Color(96, 96, 191));
    pieplot.setSectionPaint("Four", new Color(64, 64, 159));
    pieplot.setSectionPaint("Five", new Color(32, 32, 127));
    pieplot.setSectionPaint("Six", new Color(0, 0, 111));
    pieplot.setNoDataMessage("No data available");
    pieplot.setExplodePercent("Two", 0.5);
    pieplot.setLabelGenerator(new StandardPieSectionLabelGenerat
or("{0} ({2} percent)"));
    pieplot.setLabelBackgroundPaint(new Color(220, 220, 220));
    pieplot.setLegendLabelToolTipGenerator(new StandardPieSectio
nLabelGenerator("Tooltip for legend item {0}"));
    return jfreechart;
}
```

程序代码说明：

- `setSectionPaint("One", new Color(160, 160, 255))`：设置某个片区的填充颜色。第一个参数为片区的标识，第二个参数为色值。
- `setNoDataMessage("No data available")`：设置dataset为null时显示的提示信息。
- `setLabelGenerator(new StandardPieSectionLabelGenerator("{0}({2}`：设置标签显示的格式。
- `setLabelBackgroundPaint(new Color(220, 220, 220))`：设置标签的背景颜色。
- `setLegendLabelToolTipGenerator(new StandardPieSectionLabelGenera`

：设置鼠标滑过图表是显示鼠标当前片区的提示信息。
- `pieplot.setExplodePercent("Two", 0.5)`：将第2个片区取出显示。后面一个参数是取出的距离，是一个比例数。

5.10.4 类PieChartDemo3.java

功能：

显示了dataset为空时，设置的提示信息。

效果：



代码：

```
private static JFreeChart createChart(PieDataset piedataset) {
    JFreeChart jfreechart = ChartFactory.createPieChart("Pie Cha
rt Demo 3",
    null, true, true, false);
    PiePlot pieplot = (PiePlot) jfreechart.getPlot();
    pieplot.setNoDataMessage("没有有效的数据显示!");
    pieplot.setNoDataMessageFont(new Font("黑体", 2, 20));
    pieplot.setNoDataMessagePaint(Color.red);
    return jfreechart;
}
```

程序代码说明：

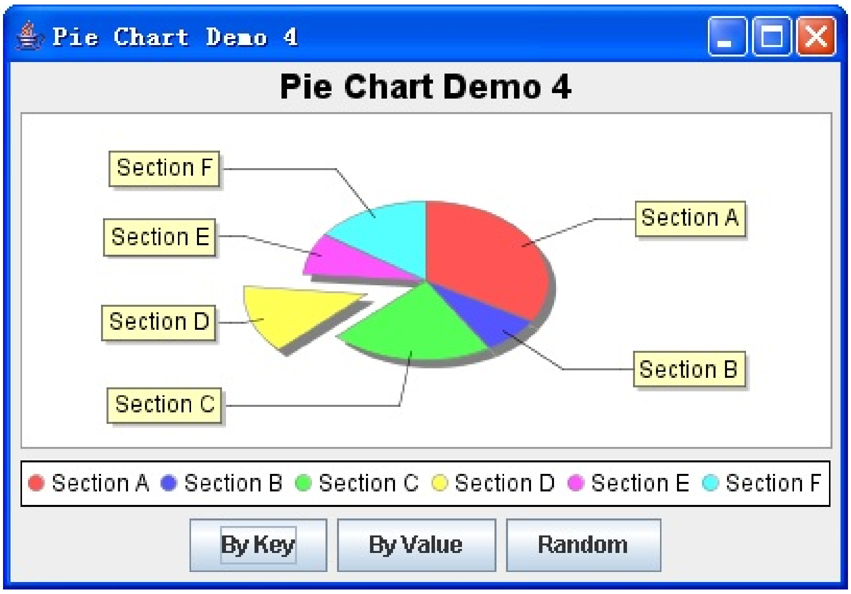
- `setNoDataMessage("没有有效的数据显示!")`：设置提示信息内容。
- `setNoDataMessageFont(new Font("Serif", 2, 10))`：设置提示信息的字体和大小。
- `setNoDataMessagePaint(Color.red)`：设置提示信息字体的颜色。

5.10.5 类PieChartDemo4.java

功能：

带有按钮的图表，通过对dataset的排序，可以改变片区的位置。

效果：



代码：

```
public void actionPerformed(ActionEvent actionevent) {
    String string = actionevent.getActionCommand();
    if ("BY_KEY".equals(string)) {
        if (!ascendingByKey) {
            dataset.sortByKeys(SortOrder.ASCENDING);
            ascendingByKey = true;
        } else {
            dataset.sortByKeys(SortOrder.DESCENDING);
            ascendingByKey = false;
        }
    } else if ("BY_VALUE".equals(string)) {
        if (!ascendingByValue) {
            dataset.sortByValues(SortOrder.ASCENDING);
            ascendingByValue = true;
        } else {
            dataset.sortByValues(SortOrder.DESCENDING);
            ascendingByValue = false;
        }
    } else if ("RANDOM".equals(string)) {
        ArrayList arraylist = new ArrayList(dataset.getKeys());
        Collections.shuffle(arraylist);
        DefaultPieDataset defaultpiedataset = new DefaultPieData
set();
        Iterator iterator = arraylist.iterator();
        while (iterator.hasNext()) {
            Comparable comparable = (Comparable) iterator.next()
;
            defaultpiedataset.setValue(comparable, dataset
            .getValue(comparable));
        }
        PiePlot pieplot = (PiePlot) chart.getPlot();
        pieplot.setDataset(defaultpiedataset);
        dataset = defaultpiedataset;
    }
}
```

程序代码说明：

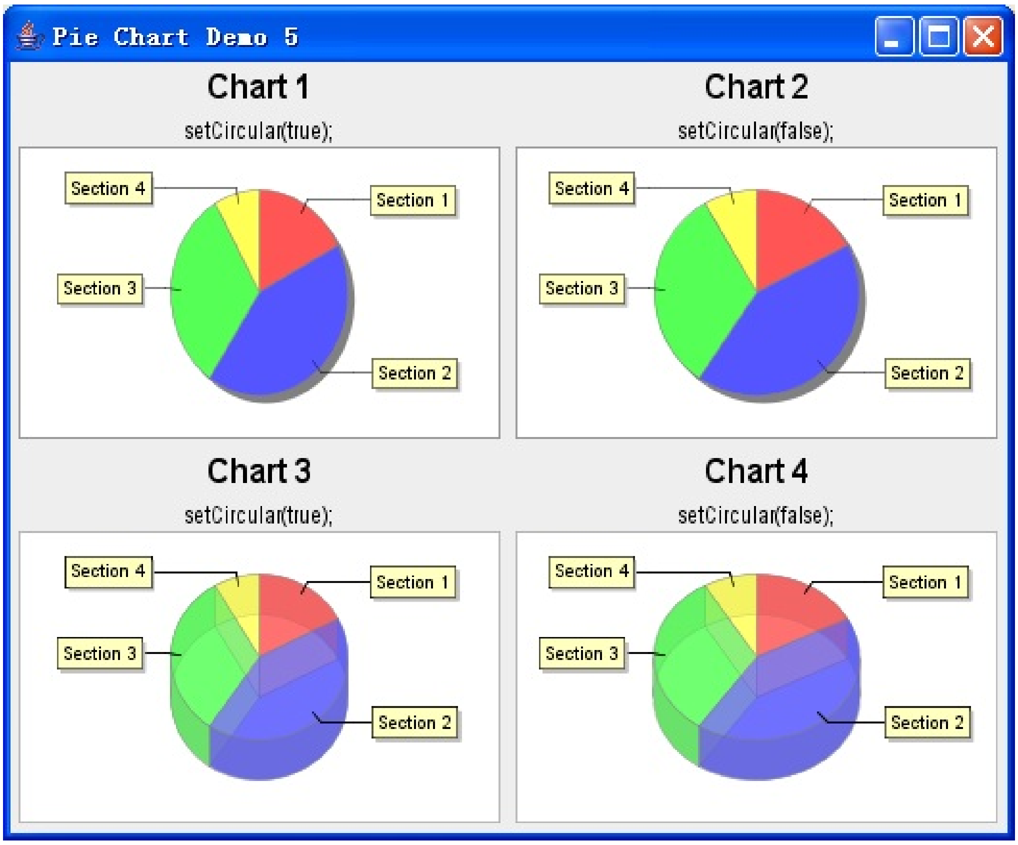
- `dataset.sortByKeys(SortOrder.ASCENDING);` 通过片区的关键值进行升序排序。
- `dataset.sortByValues(SortOrder.DESCENDING);` 通过片区的值进行降序排序。

5.10.6 类PieChartDemo5.java

功能：

右边两个饼图为原形的，而左边两个为椭圆形的。

效果：



代码：

```
public static JPanel createDemoPanel() {
    JPanel jpanel = new JPanel(new GridLayout(2, 2));
    DefaultPieDataset defaultpiedataset = new DefaultPieDataset(
);
    defaultpiedataset.setValue("Section 1", 23.3);
    defaultpiedataset.setValue("Section 2", 56.5);
    defaultpiedataset.setValue("Section 3", 43.3);
    defaultpiedataset.setValue("Section 4", 11.1);
    //
    JFreeChart jfreechart = ChartFactory.createPieChart("Chart 1
",
        defaultpiedataset, false, false, false);
    jfreechart.addSubtitle(new TextTitle("setCircular(true);", n
ew Font(
        "Dialog", 0, 12)));
    PiePlot pieplot = (PiePlot) jfreechart.getPlot();
    pieplot.setCircular(true);
    //
    JFreeChart jfreechart_0_ = ChartFactory.createPieChart("Char
t 2",
        defaultpiedataset, false, false, false);
    jfreechart_0_.addSubtitle(new TextTitle("setCircular(false);
",
        new Font("Dialog", 0, 12)));
    PiePlot pieplot_1_ = (PiePlot) jfreechart_0_.getPlot();
    pieplot_1_.setCircular(false);
    //
```

```
    JFreeChart jfreechart_2_ = ChartFactory.createPieChart3D("Ch
art 3",
        defaultpiedataset, false, false, false);
    jfreechart_2_.addSubtitle(new TextTitle("setCircular(true);"
, new Font(
        "Dialog", 0, 12)));
    PiePlot3D pieplot3d = (PiePlot3D) jfreechart_2_.getPlot();
    pieplot3d.setForegroundAlpha(0.6F);
    pieplot3d.setCircular(true);
    //
    JFreeChart jfreechart_3_ = ChartFactory.createPieChart3D("Ch
art 4",
        defaultpiedataset, false, false, false);
    jfreechart_3_.addSubtitle(new TextTitle("setCircular(false);
",
        new Font("Dialog", 0, 12)));
    PiePlot3D pieplot3d_4_ = (PiePlot3D) jfreechart_3_.getPlot()
;
    pieplot3d_4_.setForegroundAlpha(0.6F);
    pieplot3d_4_.setCircular(false);
    jpanel.add(new ChartPanel(jfreechart));
    jpanel.add(new ChartPanel(jfreechart_0_));
    jpanel.add(new ChartPanel(jfreechart_2_));
    jpanel.add(new ChartPanel(jfreechart_3_));
    jpanel.setPreferredSize(new Dimension(800, 600));
    return jpanel;
}
```

程序代码说明：

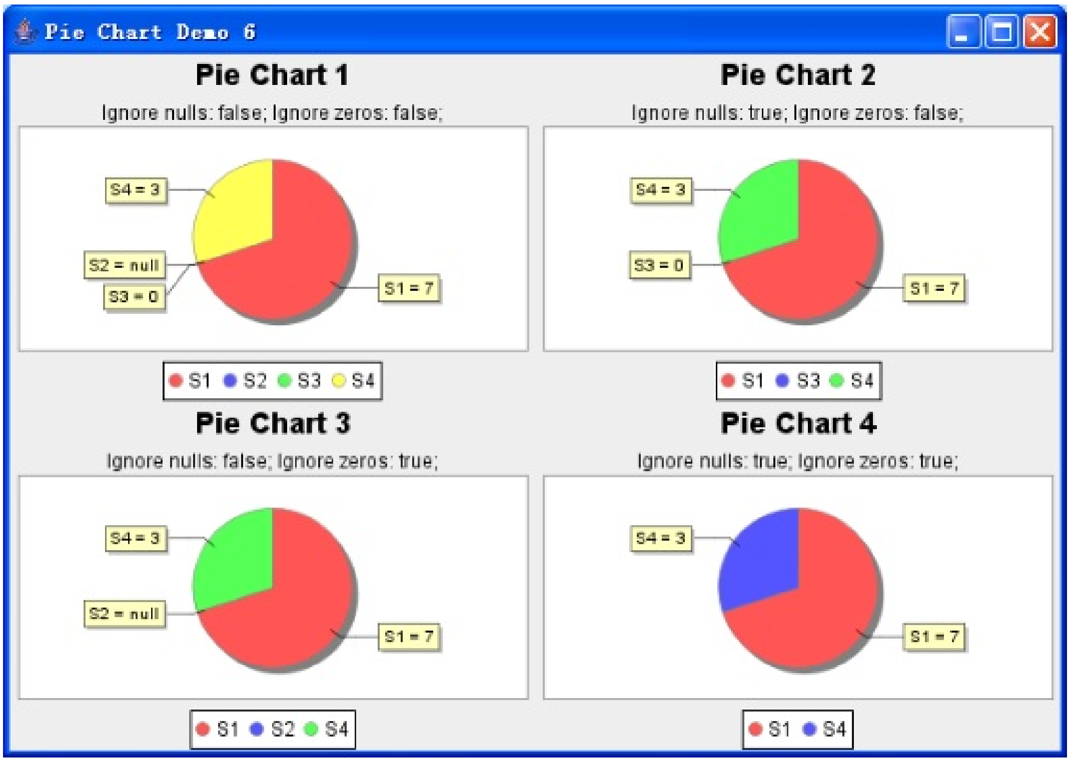
- `pieplot.setCircular(true)`：设置饼图为圆形。
- `jpanel.add` 方法可以添加多个图形的panel

5.10.7 类PieChartDemo6.java

功能：

显示饼图对零值和null值的处理。

效果：


Pie Chart Demo 6
Pie Chart 1
Ignore nulls: false; Ignore zeros: false;
S4 = 3
S2 = null
S3 = 0
S1 = 7
S1 S2 S3 S4
Pie Chart 2
Ignore nulls: true; Ignore zeros: false;
S4 = 3
S3 = 0
S1 = 7
S1 S3 S4
Pie Chart 3
Ignore nulls: false; Ignore zeros: true;
S4 = 3
S2 = null
S1 = 7
S1 S2 S4
Pie Chart 4
Ignore nulls: true; Ignore zeros: true;
S4 = 3
S1 = 7
S1 S4

代码：

```
public static JPanel createDemoPanel() {
JPanel jpanel = new JPanel(new GridLayout(2, 2));
    JFreeChart jfreechart = createChart("Pie Chart 1", createDat
aset());
    Font font = new Font("Dialog", 0, 12);
    jfreechart.addSubtitle(new TextTitle(
        "Ignore nulls: false; Ignore zeros: false;", font));
    JFreeChart jfreechart_0_ = createChart("Pie Chart 2", create
Dataset());
    jfreechart_0_.addSubtitle(new TextTitle(
        "Ignore nulls: true; Ignore zeros: false;", font));
    PiePlot pieplot = (PiePlot) jfreechart_0_.getPlot();
    pieplot.setIgnoreNullValues(true);
    pieplot.setIgnoreZeroValues(false);
    JFreeChart jfreechart_1_ = createChart("Pie Chart 3", create
Dataset());
    jfreechart_1_.addSubtitle(new TextTitle(
        "Ignore nulls: false; Ignore zeros: true;", font));
    PiePlot pieplot_2_ = (PiePlot) jfreechart_1_.getPlot();
    pieplot_2_.setIgnoreNullValues(false);
    pieplot_2_.setIgnoreZeroValues(true);
    JFreeChart jfreechart_3_ = createChart("Pie Chart 4", create
Dataset());
    jfreechart_3_.addSubtitle(new TextTitle(
        "Ignore nulls: true; Ignore zeros: true;", font));
    PiePlot pieplot_4_ = (PiePlot) jfreechart_3_.getPlot();
    pieplot_4_.setIgnoreNullValues(true);
    pieplot_4_.setIgnoreZeroValues(true);
    jpanel.add(new ChartPanel(jfreechart));
    jpanel.add(new ChartPanel(jfreechart_0_));
    jpanel.add(new ChartPanel(jfreechart_1_));
    jpanel.add(new ChartPanel(jfreechart_3_));
    return jpanel;
}
```

程序代码说明：

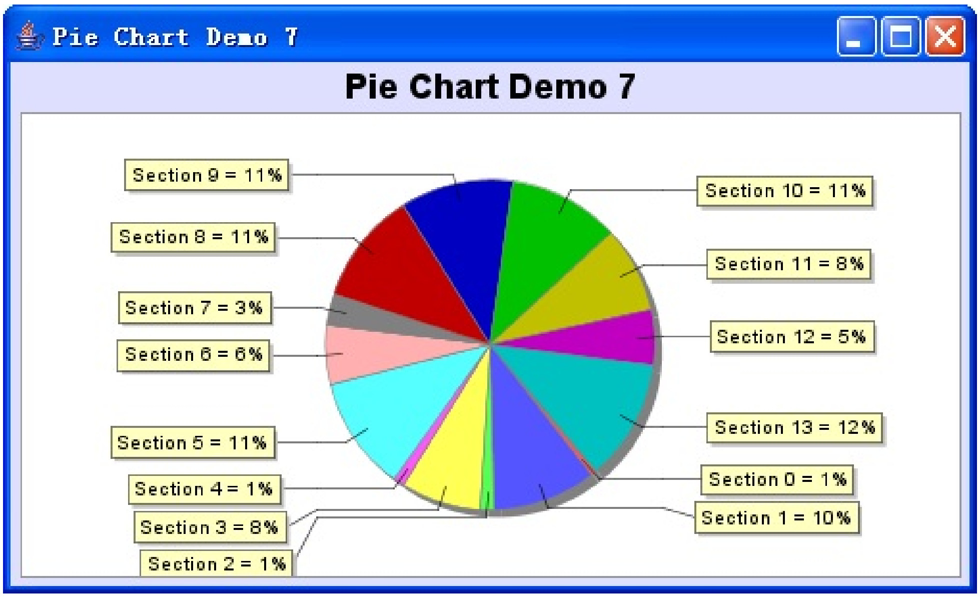
- `pieplot.setIgnoreNullValues(true)` ：设置饼图忽略null值，即是null值将不显示。
- `pieplot.setIgnoreZeroValues(false);` 设置饼图不忽略零值。即图表中显示出零值。

5.10.8 类PieChartDemo7.java

功能：

图表能够旋转，标签也随之移动。

效果：



代码：

```
public static JPanel createDemoPanel() {
    PieDataset piedataset = createDataset(14);
    JFreeChart jfreechart = ChartFactory.createPieChart("Pie Cha
rt Demo 7",
        piedataset, false, true, false);
    jfreechart.setBackgroundPaint(new Color(222, 222, 255));
    PiePlot pieplot = (PiePlot) jfreechart.getPlot();
    pieplot.setBackgroundPaint(Color.white);
    pieplot.setCircular(true);
    pieplot.setLabelGenerator(new StandardPieSectionLabelGenerat
or(
        "{0} = {2}", NumberFormat.getNumberInstance(), NumberFor
mat
        .getPercentInstance()));
    pieplot.setNoDataMessage("No data available");
    ChartPanel chartpanel = new ChartPanel(jfreechart);
    chartpanel.setPreferredSize(new Dimension(500, 270));
    Rotator rotator = new Rotator(pieplot);
    rotator.start();
    return chartpanel;
}
```

程序代码说明：

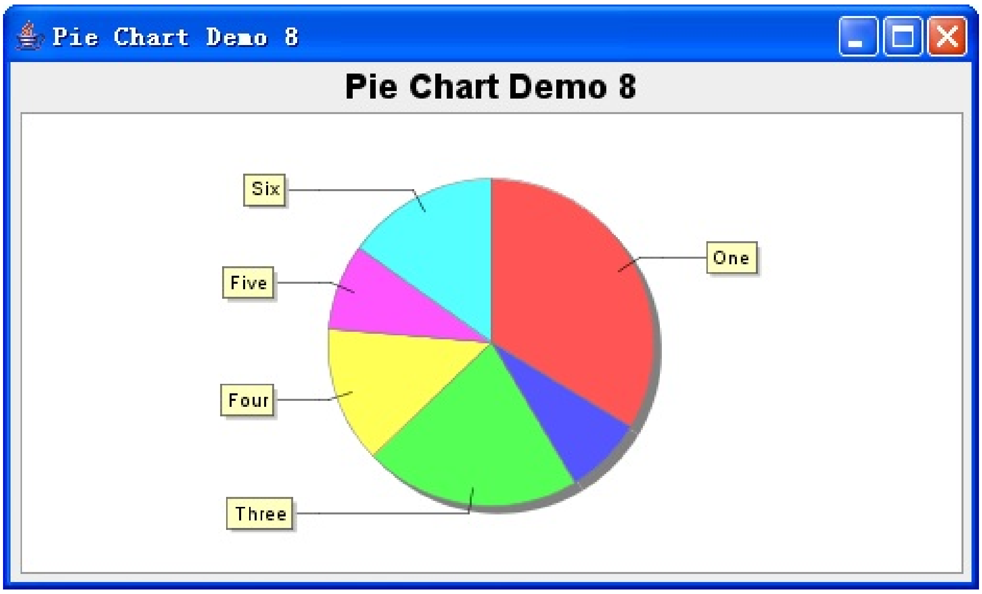
- 使用Rotator对象，旋转饼图。代码如上。

5.10.9 类PieChartDemo8.java

功能：

编写自定义标签产生器，将Two标签不显示。

效果：



代码：

```
private static JFreeChart createChart(PieDataset piedataset) {
    JFreeChart jfreechart = ChartFactory.createPieChart("Pie Cha
rt Demo 8",
    piedataset, false, true, false);
    PiePlot pieplot = (PiePlot) jfreechart.getPlot();
    pieplot.setLabelGenerator(new CustomLabelGenerator());
    return jfreechart;
}
static class CustomLabelGenerator implements PieSectionLabelGene
rator {
    public String generateSectionLabel(PieDataset piedataset,
    Comparable comparable) {
    String string = null;
        if (piedataset != null && !comparable.equals("Two"))
            string = comparable.toString();
        return string;
    }
    public AttributedString generateAttributedSectionLabel(
        PieDataset piedataset, Comparable comparable) {
        Object object = null;
        String string = comparable.toString();
        String string_0_ = (string + " : " + String.valueOf(pied
ataset
            .getValue(comparable)));
        AttributedString attributedstring = new AttributedString
(string_0_);
        attributedstring.addAttribute(TextAttribute.WEIGHT,
        TextAttribute.WEIGHT_BOLD, 0, string.length() - 1);
        return attributedstring;
    }
}
```

程序代码说明：

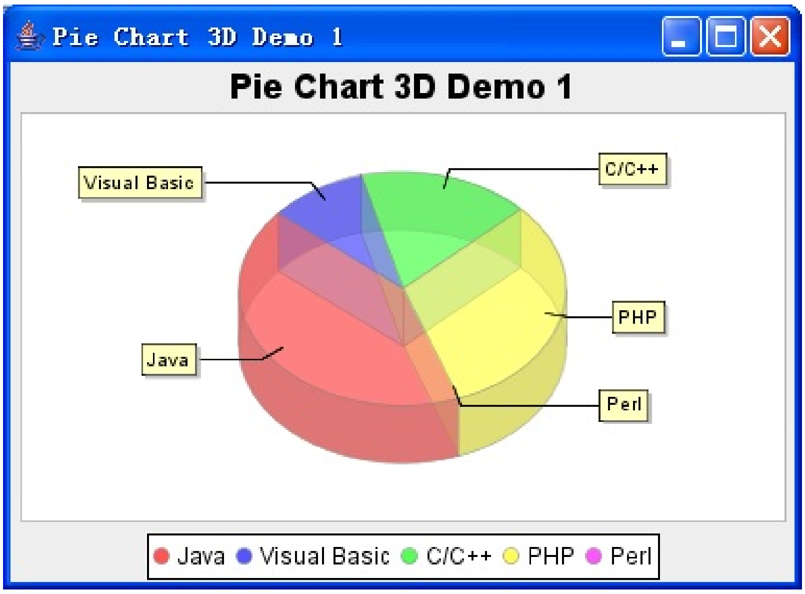
- 自定义 `CustomLabelGenerator` ，必须实现接口 `PieSectionLabelGenerator` 。

5.10.10 类PieChart3DDemo1.java

功能：

3D效果的饼图。

效果：



代码：

```
private static JFreeChart createChart(PieDataset piedataset) {
    JFreeChart jfreechart = ChartFactory.createPieChart3D(
        "Pie Chart 3D Demo 1", piedataset, true, true, false);
    PiePlot3D pieplot3d = (PiePlot3D) jfreechart.getPlot();
    pieplot3d.setStartAngle(180.0);
    pieplot3d.setDirection(Rotation.CLOCKWISE);
    pieplot3d.setForegroundAlpha(0.5F);
    pieplot3d.setNoDataMessage("No data to display");
    return jfreechart;
}
```

程序代码说明：

- 使用 `ChartFactory` 的方法 `createPieChart3D` 创建3D效果的饼图。
- `setStartAngle(180.0)` ：设置旋转角度。
- `setDirection(Rotation.CLOCKWISE)` ：设置旋转方向， `Rotation.CLOCKWISE` 为顺时针。
- `setForegroundAlpha(0.5F)` ：设置图表透明图0.0~1.0范围。0.0为完全透

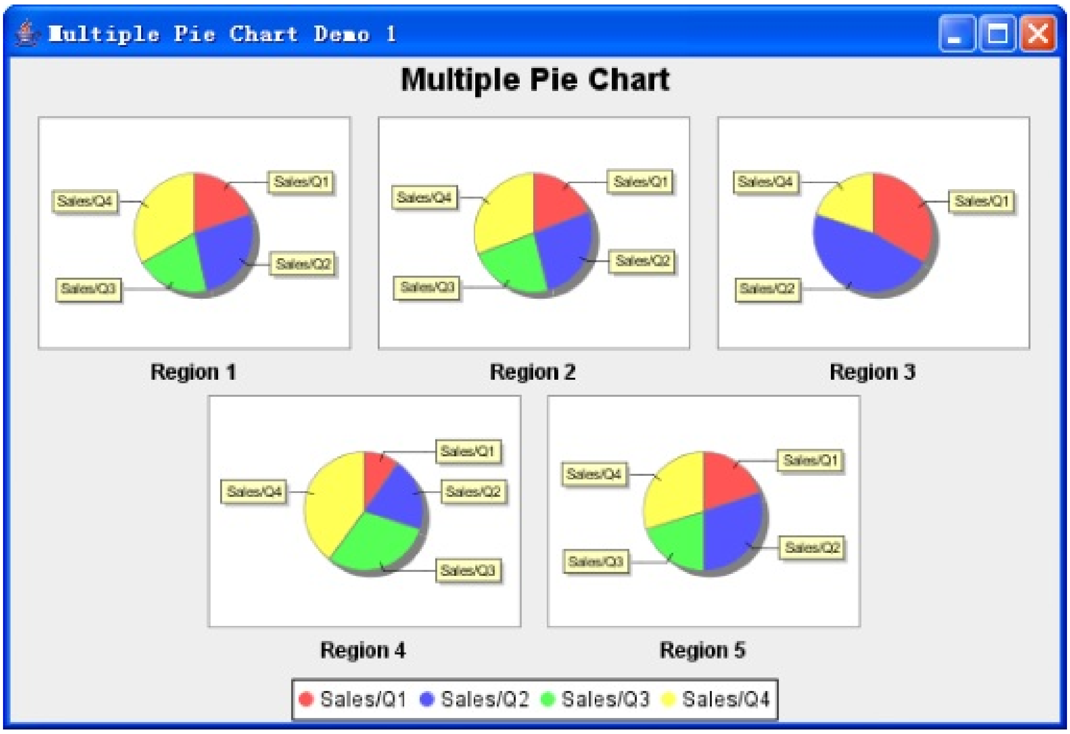
明，1.0为完全不透明。

5.10.11 类MultiplePieChartDemo1.java

功能：

使用CategoryDataset数据集，在一个图表上产生多个饼图。

效果：



代码：

```
private static CategoryDataset createDataset() {
    double[][] ds = { { 3.0, 4.0, 3.0, 5.0 }, { 5.0, 7.0, 6.0, 8
.0 },
        { 5.0, 7.0, Double.NaN, 3.0 }, { 1.0, 2.0, 3.0, 4.0 },
        { 2.0, 3.0, 2.0, 3.0 } };
    CategoryDataset categorydataset = DatasetUtilities
        .createCategoryDataset("Region ", "Sales/Q", ds);
    return categorydataset;
}
```

程序代码说明：

- 创建 `CategoryDataset` 的方法。

```
private static JFreeChart createChart(CategoryDataset categoryda
taset) {
    JFreeChart jfreechart = ChartFactory.createMultiplePieChart(
        "Multiple Pie Chart", categorydataset, TableOrder.BY_ROW
, true, true, false);
    MultiplePiePlot multiplepieplot = (MultiplePiePlot) jfreecha
rt .getPlot();
    JFreeChart jfreechart_0_ = multiplepieplot.getPieChart();
    PiePlot pieplot = (PiePlot) jfreechart_0_.getPlot();
    pieplot.setLabelGenerator(new StandardPieSectionLabelGenerat
or("{0}"));
    pieplot.setLabelFont(new Font("SansSerif", 0, 8));
    pieplot.setInteriorGap(0.3);
    return jfreechart;
}
```

程序代码说明：

- 使用 `ChartFactory` 的方法 `createMultiplePieChart()` 创建多个饼图的图表。
- `multiplepieplot.getPieChart()`：获得单个饼图的图表。

6 直方条形图（Bar Charts）

6.1 简介

本章详细介绍了使用JFreeChart创建直方条形图的过程。开始我们先用一个简单的直方条形图例子进行说明，然后进一步深入了解JFreeChart为直方条形图提供的定制方法。在了解了这些标准的直方条形图配置项之后，然后再深入了解更复杂的图表：

- 堆栈式直方条形图
- 时序数据的条形直方图
- 柱状图

本章结束之后，我们将会对JFreeChart支持直方条形图创建的特点有个整体的了解。

6.2 创建一个直方条形图

6.2.1 概述

直方条形图常常被用来显示表列数据。如下表，为一个简单的两行、三列数据。

表 6.1

	Colnums1	Colnums2	Colnums3
Row1	1.0	5.0	3.0
Row2	2.0	3.0	2.0

在JFreeChart里，这个表格数据封装为一个dataset数据对象，每列标题为一个种类，每行为一个系列。每行标题为一个系列名称（或者系列关键值）。直方条形图展现的数据图如图6.2.

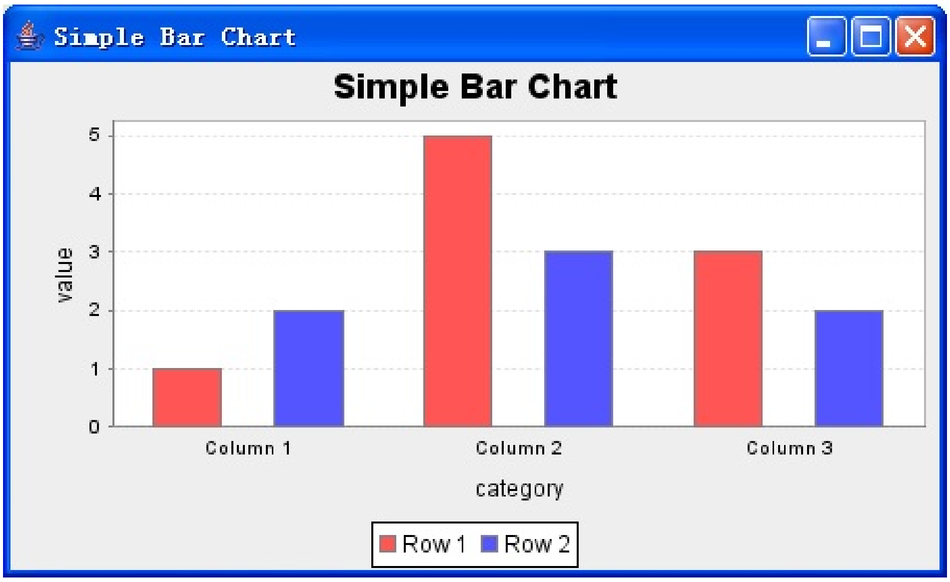


图6.2 简单的直方条形图（参见：BarExample1.java）

在这个图表的实例中，我们可以看到JFreeChart将每列数据（即一个种类）组合在一起。而且对每行数据（即每个系列）使用各种颜色高亮显示。图表的图例将颜色和系列的名称/关键值对应起来。

6.2.2 创一个dataset

创建直方条形图的第一步就是创建一个合适的dataset数据集。JFreeChart为直方条形图提供的访问表列数据的一系列方法，必须符合接口CategoryDataset定义。

JFreeChart中提供了一个便利的实现CategoryDataset接口的类为DefaultCategoryDataset。下面显示我们如何使用这个类来封装表6.1数据。代码如下：

```
private CategoryDataset createDataset() {
    DefaultCategoryDataset dataset = new DefaultCategoryDataset(
);
    dataset.addValue(1.0, "Row 1", "Column 1");
    dataset.addValue(5.0, "Row 1", "Column 2");
    dataset.addValue(3.0, "Row 1", "Column 3");
    dataset.addValue(2.0, "Row 2", "Column 1");
    dataset.addValue(3.0, "Row 2", "Column 2");
    dataset.addValue(2.0, "Row 2", "Column 3");
    return dataset;
}
```

6.2.3 创建一个chart图表

接下来就是要创建一个JFreeChart的实例，使用上面提供的dataset数据集画一个直方条形图。简单的，我们使用ChartFactory类来创建这个JFreeChart实例。代码如下：

```
private JFreeChart createChart(CategoryDataset dataset) {
    JFreeChart chart = ChartFactory.createBarChart(
        "BarChartDemo", // chart title
        "Category", // domain axis label
        "Value", // range axis label
        dataset, // data
        PlotOrientation.VERTICAL, // orientation
        true, // include legend
        true, // tooltips?
        false // URLs?
    );
    return chart;
}
```

CreateBarChart()的大部分参数是比较容易理解的，但其中一部分还需要进一步说明。

- 图显示的方向可以是水平的，还是可以是垂直的。
- 图表的信息提示，是否要添加，有一个标志来控制——在上面的例子中，我们将这个标识设置为true，因此当我们在一个swing应用窗口显示这个图表时，我们会看到这个信息提示。
- URLs标志，设置为false。

我们完成这个直方条形图后，我们将会过头来，仔细看看ChartFactory类在后台做了写什么。

6.2.4 显示该chart图表

为了完成我们的第一个直方条形图实例，我们将JFreeChart实例传给一个ChartPanel对象，然后在一个Swing应用窗口上显示该实例。全部的代码如下：

```
import java.awt.Dimension;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;
/**
* A simple demonstration application showing how to create a bar
 chart.
*/
public class BarExample1 extends ApplicationFrame {
    /**
    * Creates a new demo instance.
    *
    * @param title
    * the frame title.
    */
    public BarExample1(String title) {
        super(title);
        DefaultCategoryDataset dataset = new DefaultCategoryData
set();
        dataset.addValue(1.0, "Row 1", "Column 1");
        dataset.addValue(5.0, "Row 1", "Column 2");
        dataset.addValue(3.0, "Row 1", "Column 3");
        dataset.addValue(2.0, "Row 2", "Column 1");
        dataset.addValue(3.0, "Row 2", "Column 2");
        dataset.addValue(2.0, "Row 2", "Column 3");
        JFreeChart chart = ChartFactory.createBarChart(
            "Bar Chart Demo", // chart
            // title
            "Category", // domain axis label
            "Value", // range axis label
            dataset, // data
            PlotOrientation.VERTICAL, // orientation
            true, // include legend
            true, // tooltips?
            false // URLs?
        );
        ChartPanel chartPanel = new ChartPanel(chart, false);
        chartPanel.setPreferredSize(new Dimension(500, 270));
        setContentPane(chartPanel);
    }
    /**
    * Starting point for the demonstration application.
    *
    * @param args
    * ignored.
```

```
    */
    public static void main(String[] args) {
        BarExample1 demo = new BarExample1("Bar Demo 1");
        demo.pack();
        RefineryUtilities.centerFrameOnScreen(demo);
        demo.setVisible(true);
    }
}
```

完成这些代码后，运行代码，将会显示如6.2图的界面。

6.3 ChartFactory类

在上面的实例代码中，我们使用ChartFactory类来组装一个JFreeChart实例来显示一个直方条形图。下面我们更仔细的看一下该类是如何工作的，因此我们可以看到直方条形图更多底层的框架结构。理解底层结构的关键是能定制图表的外观。下面是ChartFactory方法createBarChart()方法部分代码：

```
CategoryAxis categoryAxis = new CategoryAxis(categoryAxisLabel);
ValueAxis valueAxis = new NumberAxis(valueAxisLabel);
BarRenderer renderer = new BarRenderer();
……
CategoryPlot plot = new CategoryPlot(dataset, categoryAxis, valu
eAxis,
renderer);
plot.setOrientation(orientation);
JFreeChart chart = new JFreeChart(title, JFreeChart.DEFAULT_TITL
E_FONT, plot, legend);
```

以下就是代码所做的工作。

- 我们的直方条形图有两个轴，一个轴显示dataset（CategoryAxis）的种类，另一个是显示带有数据（NumberAxis）刻度的数据轴。上面代码中代码1、2行建立了这两个轴，轴的标签是createBarChart()方法传入的。
- 第三行，创建了一个BarRender——该类为每一个数据项目画直方图。该render处理大部分画图工作，我们后续代码也会看到可以使用另一个类型的render替换现有的render，来改变图表的整个外观。
- Dataset、axes和render都由CategorryPlot来管理，CategorryPlot系统组件之间的大部分交互工作。当我们定制一个图表时，我们经常需要先获得整个图表plot、renderer和dataset的引用。在代码的第四行，创建了一个plot，然后其他组件对它进行赋值。
- 最后，在JFreeChart实例中，这个plot用指定的标题被封装。JFreeChart类提供了比较高层次的访问图表。但在这个plot曾思图表就大部分被定义出来了（Plot管理很多对象，例如axes、dataset和renderer）。

图表的内部结构基本上是由上面的知识理论组成。在后续的章节，我们会逐渐学习更多的定制我们图表的方法。

6.4 直方条形图的简单定制

调用JFreeChart和CategoryPlot类方法可以进行一些简单的直方图表外观的修改。例如，改变图表和区域的背景颜色代码如下：

```
chart.setBackgroundPaint(Color.white);
CategoryPlot plot = (CategoryPlot) chart.getPlot();
plot.setBackgroundPaint(Color.lightGray);
plot.setRangeGridlinePaint(Color.white);
```

该片段代码（摘自BarExample2.java类）显示了改变图表的背景颜色、获得图表的plot（区域）的引用，并且进行了修改——效果如图6.3.

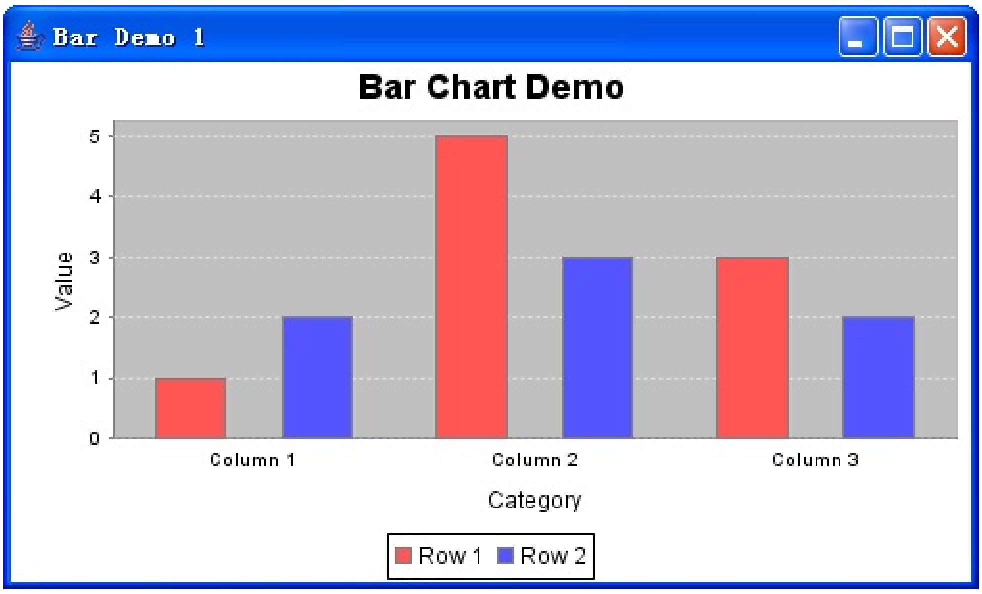


图6.3一个直方条形图（参考：BarExample2.java）

区域Plot的引用（CategoryPlot）是必须的——转换类型也是非常安全的，因为我们知道该图表类型使用CategoryPlot。JFreeChart使用不同的区域类型（比如PiePlot、XYPlot）控制不同类型的图表。我们必须将plot的引用转化成图表响应的类型，因为基本类Plot仅仅定义了一些通用的属性和方法。随着对JFreeChart了解的加深，我们将学习每一种图表使用的不同的plot子类。

在我们的例子中，我们使用plot的引用来改变水平轴的网格线颜色。看一下CategoryPlot类的API文件，就会看到我们能够修改的地方。

6.5 定制外观

回顾6.3节内容，CategoryPlot管理这一个BarRenderer的实例renderer。如果我们想获得这个renderer的引用，大量的定制选择项会变得有效。

6.5.1 直方条形图颜色

改变图表中每个系列直方图的颜色，使用如下代码：

```
BarRenderer renderer = (BarRenderer) plot.getRenderer();
renderer.setSeriesPaint(0, Color.gray);
renderer.setSeriesPaint(1, Color.orange);
renderer.setDrawBarOutline(false);
```

运行上面代码显示的结果如下图6.4.注意setSeriesPaint（）方法是在抽象AbstractRenderer基类里面定义的——所以，我们可以在任何类型的renderer里面使用。

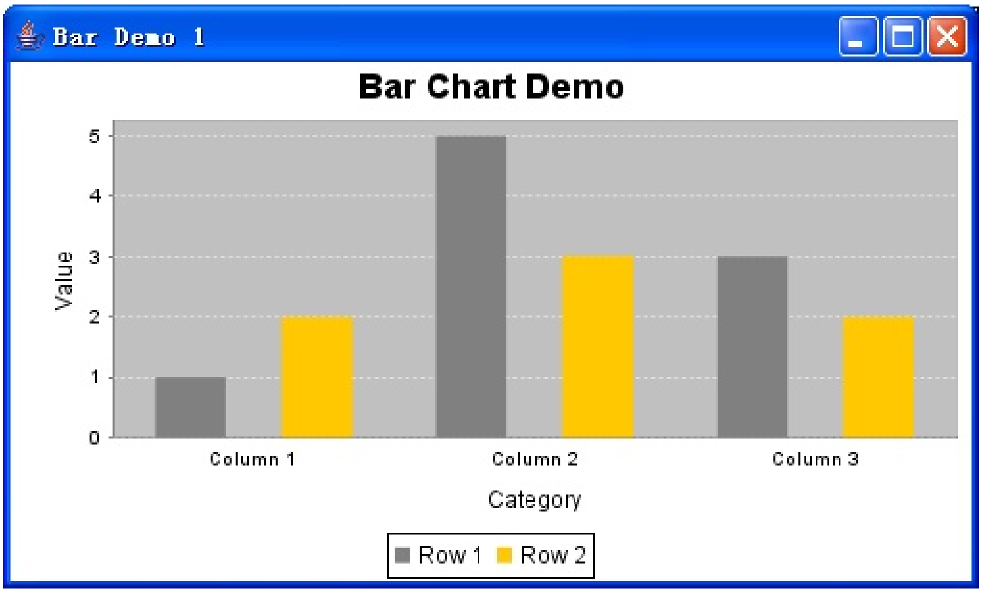


图6.4一个直方条形图（参考：BarExample3.java）

6.5.2 种类里直方条形图之间的空间

此外，renderer还可以控制每个种类中直方条形图之间的间距。因此我们可以在同一个种类中将空间完全去掉，代码如下：

```
BarRenderer renderer = (BarRenderer) plot.getRenderer();
renderer.setItemMargin(0.0);
```

代码显示的结果如图6.5所示。

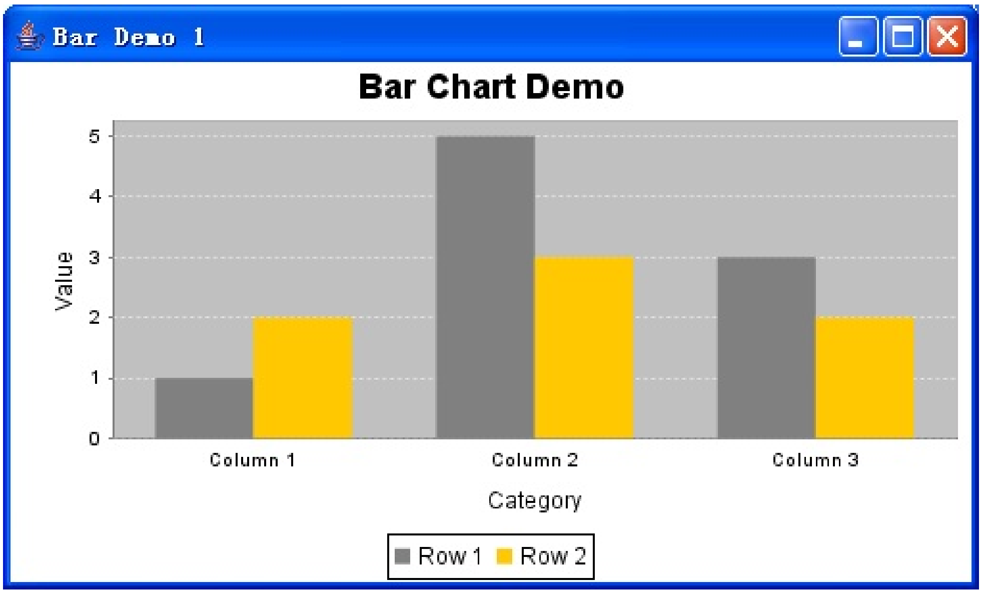


图6.5一个直方条形图（参考：BarExample4.java）

注意条形图看上去有点变宽——主要是因为JFreeChart分配空间时，分配给种类条形图之间的间距的尺度比较少，所以看上去就显得有点长宽了。

6.6 示例代码解读

6.6.1 体会

与饼图的数据集不同之处在于：

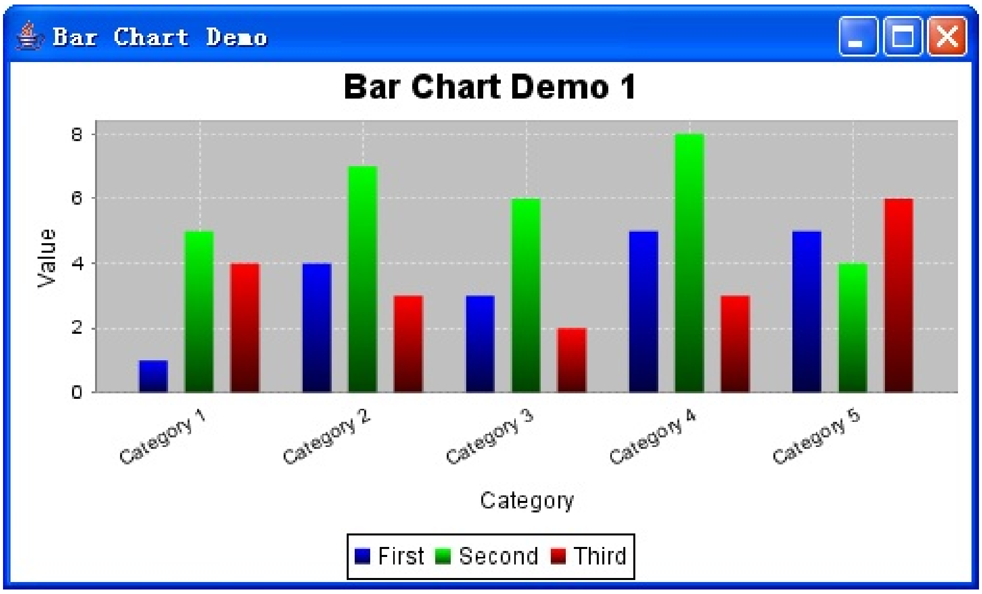
饼图数据集是key/value二维数据（PieDataset）。而直方条形图需要三维数据（CategoryDataset）。

6.6.2 类BarChartDemo1.java

功能：

一个简单的直方条形图。使用GradientPaint实例对象为每一个系列修改renderer

效果：



代码：

```
public class BarChartDemo1 extends ApplicationFrame {
    private static final long serialVersionUID = 1L;
    public BarChartDemo1(String string) {
        super(string);
        JPanel jpanel = createDemoPanel();
        jpanel.setPreferredSize(new Dimension(500, 270));
        setContentPane(jpanel);
    }
    private static CategoryDataset createDataset() {
        String string = "First";
        String string_0_ = "Second";
        String string_1_ = "Third";
        String string_2_ = "Category 1";
```

```
        String string_3_ = "Category 2";
        String string_4_ = "Category 3";
        String string_5_ = "Category 4";
        String string_6_ = "Category 5";
        DefaultCategoryDataset defaultcategorydataset = new Defa
ultCategoryDataset();
        defaultcategorydataset.addValue(1.0, string, string_2_);
        defaultcategorydataset.addValue(4.0, string, string_3_);
        defaultcategorydataset.addValue(3.0, string, string_4_);
        defaultcategorydataset.addValue(5.0, string, string_5_);
        defaultcategorydataset.addValue(5.0, string, string_6_);
        defaultcategorydataset.addValue(5.0, string_0_, string_2
_);
        defaultcategorydataset.addValue(7.0, string_0_, string_3
_);
        defaultcategorydataset.addValue(6.0, string_0_, string_4
_);
        defaultcategorydataset.addValue(8.0, string_0_, string_5
_);
        defaultcategorydataset.addValue(4.0, string_0_, string_6
_);
        defaultcategorydataset.addValue(4.0, string_1_, string_2
_);
        defaultcategorydataset.addValue(3.0, string_1_, string_3
_);
        defaultcategorydataset.addValue(2.0, string_1_, string_4
_);
        defaultcategorydataset.addValue(3.0, string_1_, string_5
_);
        defaultcategorydataset.addValue(6.0, string_1_, string_6
_);
        return defaultcategorydataset;
    }
    private static JFreeChart createChart(CategoryDataset catego
rydataset) {
        JFreeChart jfreechart = ChartFactory.createBarChart("Bar
 Chart Demo 1",
            "Category", "Value", categorydataset, PlotOrientatio
n.VERTICAL,
            true, true, false);
        jfreechart.setBackgroundPaint(Color.WHITE);
        CategoryPlot categoryplot = (CategoryPlot) jfreechart.ge
tPlot();
        categoryplot.setBackgroundPaint(Color.lightGray);
        categoryplot.setDomainGridlinePaint(Color.white);
        categoryplot.setDomainGridlinesVisible(true);
        categoryplot.setRangeGridlinePaint(Color.white);
        //刻度轴刻度设置
        NumberAxis numberaxis = (NumberAxis) categoryplot.getRan
geAxis();
        numberaxis.setStandardTickUnits(NumberAxis.createInteger
TickUnits());
        //renderer设置
```

```
        BarRenderer barrenderer = (BarRenderer) categoryplot.get
Renderer();
        barrenderer.setDrawBarOutline(false);//设置外廓线不可见
        GradientPaint gradientpaint = new GradientPaint(0.0F, 0.
0F, Color.blue,
            0.0F, 0.0F, new Color(0, 0, 64));
        GradientPaint gradientpaint_7_ = new GradientPaint(0.0F,
 0.0F,
            Color.green, 0.0F, 0.0F, new Color(0, 64, 0));
        GradientPaint gradientpaint_8_ = new GradientPaint(0.0F,
 0.0F,
            Color.red, 0.0F, 0.0F, new Color(64, 0, 0));
        barrenderer.setSeriesPaint(0, gradientpaint);
        barrenderer.setSeriesPaint(1, gradientpaint_7_);
        barrenderer.setSeriesPaint(2, gradientpaint_8_);
        //设置种类标签旋转的角度，逆时针旋转
        CategoryAxis categoryaxis = categoryplot.getDomainAxis()
;
        categoryaxis.setCategoryLabelPositions(CategoryLabelPosi
tions
            .createUpRotationLabelPositions(Math.PI / 6));
        return jfreechart;
    }
    public static JPanel createDemoPanel() {
        JFreeChart jfreechart = createChart(createDataset());
        return new ChartPanel(jfreechart);
    }
    public static void main(String[] strings) {
        BarChartDemo1 barchartdemo1 = new BarChartDemo1("Bar Cha
rt Demo");
        barchartdemo1.pack();
        RefineryUtilities.centerFrameOnScreen(barchartdemo1);
        barchartdemo1.setVisible(true);
    }
}
```

程序代码说明：

- `Main()` 方法执行直方条形图。编写方法与饼图一样。
- `BarChartDemo1` 构造函数中创了一个JPanel，并设置大小。
- `createDemoPanel()` 方法创建了一个JPanel，并且在该panel上创建了直方条形图。
- `createDataset()` 方法创建了数据集。类型为CategoryDataset。注意数据集为三维数据。与饼图不同。
- 使用 `ChartFactory.createBarChart()` 方法创建直方条形图
- `jfreechart.setBackgroundPaint(Color.WHITE)` ：设置图表的背景颜色。
- `categoryplot.setBackgroundPaint(Color.lightGray)` ：设置直方条形图的背景颜色。
- `setDomainGridlinePaint（Color.whites）` ：设置垂直格线的颜色。默认

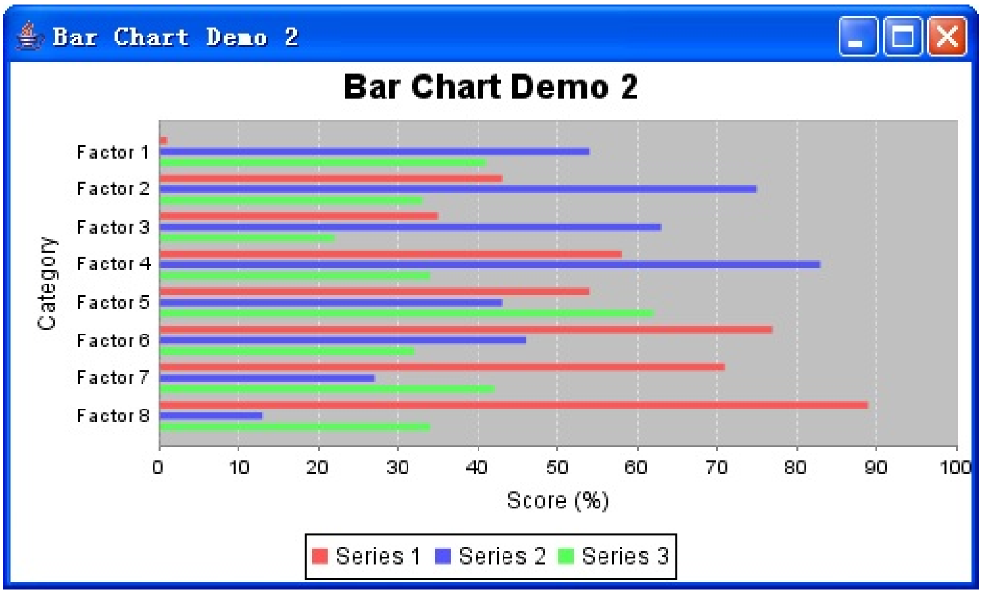
不可见。
- `setRangeGridlinePaint(Color.white)`：设置水平格线的颜色。默认可见。
- `setStandardTickUnits(NumberAxis.createIntegerTickUnits())`：设置数据轴的刻度递进范围。
- `GradientPaint` 类用来设置渐变色。
- `categoryaxis.setCategoryLabelPositions()`：设置标签文字旋转的角度。

6.6.3 类BarChartDemo2.java

功能：

显示水平的直方条形图。

效果：



代码：

```
private static CategoryDataset createDataset() {
    double[][] ds = { { 1.0, 43.0, 35.0, 58.0, 54.0, 77.0, 71.0,
 89.0 },
    { 54.0, 75.0, 63.0, 83.0, 43.0, 46.0, 27.0, 13.0 },
    { 41.0, 33.0, 22.0, 34.0, 62.0, 32.0, 42.0, 34.0 } };
    return DatasetUtilities.createCategoryDataset("Series ", "Fa
ctor ", ds);
}
private static JFreeChart createChart(CategoryDataset categoryda
taset) {
    JFreeChart jfreechart = ChartFactory.createBarChart("Bar Cha
rt Demo 2",
        "Category", "Score (%)", categorydataset,
    PlotOrientation.VERTICAL, true, true, false);
    jfreechart.setBackgroundPaint(Color.white);
    CategoryPlot categoryplot = (CategoryPlot) jfreechart.getPlo
t();
    categoryplot.setBackgroundPaint(Color.lightGray);
    categoryplot.setRangeGridlinePaint(Color.white);
    categoryplot.setRangeAxisLocation(AxisLocation.BOTTOM_OR_LEF
T);
    NumberAxis numberaxis = (NumberAxis) categoryplot.getRangeAx
is();
    numberaxis.setRange(0.0, 100.0);
    numberaxis.setStandardTickUnits(NumberAxis.createIntegerTick
Units());
    BarRenderer barrenderer = (BarRenderer) categoryplot.getRend
erer();
    barrenderer.setDrawBarOutline(false);
    barrenderer
        .setLegendItemToolTipGenerator(new StandardCategorySerie
sLabelGenerator(
        "Tooltip: {0}"));
    return jfreechart;
}
```

程序代码说明：

- 数据集的创建另一种方式，使用二维数组。

7 折线图

7.1 简介

本章讲述了JFreeChart创建折线图的内容。我们可以使用CategoryDataset或XYDataset数据集接口创建折线图。

7.2 使用categoryDataset数据集创建折线图

7.2.1 概述

使用CategoryDataset创建的折线图将每个数据点（种类，值）使用一条直线连接起来。本章讲的一个简单应用产生如下界面，如图7.1：

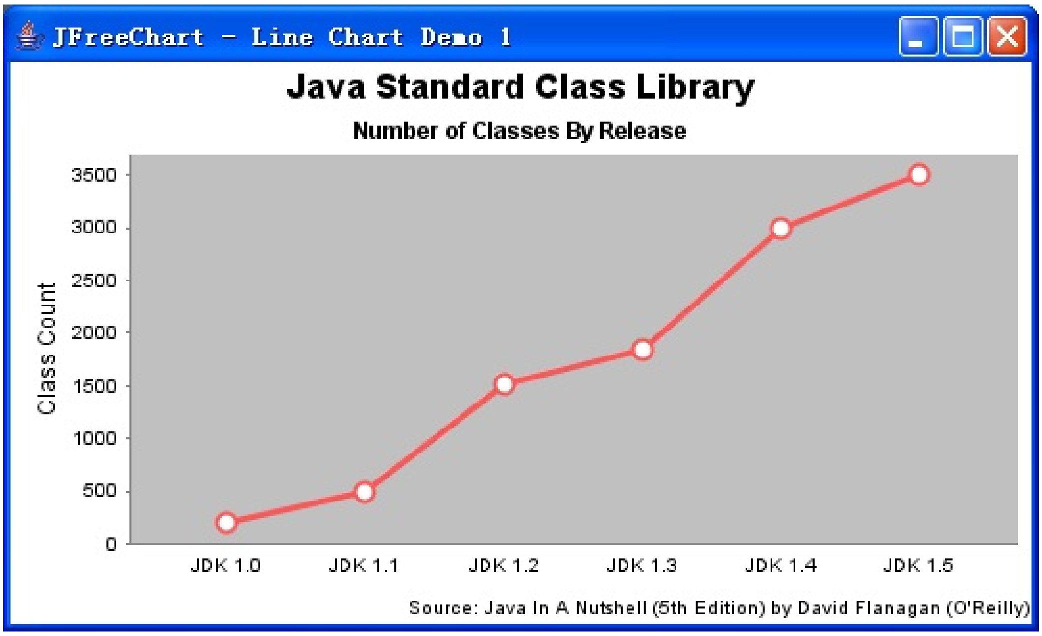


图7.1 一个简单的折线图

全部的代码简JFreeChart开发指南一并下载的demo（参考：LineChartDemo1.java）。

7.2.2 CategoryDataset

正如其他图表一样，创建折线图的第一步是创建第一个dataset。在本例子中，使用DefaultCategoryDataset，代码如下：

```
private static CategoryDataset createDataset() {
    DefaultCategoryDataset defaultcategorydataset = new DefaultC
ategoryDataset();
    defaultcategorydataset.addValue(212.0, "Classes", "JDK 1.0")
;
    defaultcategorydataset.addValue(504.0, "Classes", "JDK 1.1")
;
    defaultcategorydataset.addValue(1520.0, "Classes", "JDK 1.2"
);
    defaultcategorydataset.addValue(1842.0, "Classes", "JDK 1.3"
);
    defaultcategorydataset.addValue(2991.0, "Classes", "JDK 1.4"
);
    defaultcategorydataset.addValue(3500.0, "Classes", "JDK 1.5"
);
    return defaultcategorydataset;
}
```

注意：你可以使用任何实现Category接口的数据集。

7.2.3 创建图表

ChartFactory类提供了一个便利的方法createLineChart()创建折线图。代码如下：

```
JFreeChart jfreechart = ChartFactory.createLineChart(
    "Java Standard Class Library",// 图表标题
    null, // 主轴标签
    "Class Count",// 范围轴标签
    categorydataset, // 数据集
    PlotOrientation.VERTICAL,// 方向
    false, // 是否包含图例
    true, // 提示信息是否显示
    false// 是否使用urls
);
```

该方法构建了一个带有标题、图例、和相应的数轴和心态提示产生器的JFreeChart对象。创建Dataset数据集的过程见上节。

7.2.4 定制图表

折线图表将使用大部分缺省的属性来进行初始化。当然了，我们也可以随意修改折线图的属性，来改变我们图表的外观。在本例中，我们通过下面的方式定制折线图：

- 在图表上添加两个副标题；
- 图表的背景颜色设成白色；
- 图区背景颜色设成亮灰色；

- 网格线颜色改变成白色；
- 范围轴修改成仅显示整数数值；
- renderer使用白色填充的形状。

首先，将副标题添加在缺省的位置（主标题下方），代码如下：

```
jfreechart.addSubtitle(new TextTitle("Number of Classes By Relea
se"));
```

第二个副标题加入了一些额外的代码，来改变字体，并放置在图表的下方，并且靠右对其，代码如下：

```
TextTitle texttitle = (new TextTitle("Source: Java In A Nutshell
 (5th Edition) by David Flanagan (O'Reilly)"));
texttitle.setFont(new Font("SansSerif", 0, 10));
texttitle.setPosition(RectangleEdge.BOTTOM);
texttitle.setHorizontalAlignment(HorizontalAlignment.RIGHT);
jfreechart.addSubtitle(texttitle);
```

改变图表的背景颜色非常简单，因为JFreeChart类就有设置背景颜色的属性。代码如下：

```
//改变图表的背景颜色
jfreechart.setBackgroundPaint(Color.white);
```

如果改变其他属性的，则需要首先获得图表CategoryPlot对象的引用，然后对该引用进行相应属性的设置。获得对象引用的代码如下：

```
CategoryPlot categoryplot = (CategoryPlot) jfreechart.getPlot();
```

使用CategoryPlot设置图区的背景颜色为亮灰色，设置网格线颜色为白色的代码如下：

```
categoryplot.setBackgroundPaint(Color.lightGray);
categoryplot.setRangeGridlinePaint(Color.white);
```

图区负责在图表上画出数据和轴。其中一部分工作由renderer来完成，我们可以通过getRender()来获得一个renderer。renderer维护大部分与图表内数据项的显示相关的属性。代码如下：

```
LineAndShapeRenderer renderer = (LineAndShapeRenderer) categoryp
lot.getRenderer();
renderer.setShapesVisible(true);
renderer.setDrawOutlines(true);
renderer.setUseFillPaint(true);
```

同时图区也管理着图表所有的轴。在本实例中，修改范围轴以便范围轴的刻度标签显示为整数值。

```
NumberAxis rangeAxis = (NumberAxis) categoryplot.getRangeAxis();
rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits
());
```

定制图表还有很多其他的方式。具体的内容详见本文档相关章节，API文档以及源代码实例。

7.2.5 程序的全部代码

```
/* -------------------
* LineChartDemo1.java
* -------------------
* (C) Copyright 2002-2005, by Object Refinery Limited.
*
*/
package demo;
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.geom.Ellipse2D;
import java.net.URL;
import javax.swing.ImageIcon;
import javax.swing.JPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.renderer.category.LineAndShapeRenderer;
import org.jfree.chart.title.TextTitle;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.HorizontalAlignment;
import org.jfree.ui.RectangleEdge;
import org.jfree.ui.RefineryUtilities;
public class LineChartDemo1 extends ApplicationFrame {
```

```
    /**
    *
    */
    private static final long serialVersionUID = -63543506043130
79793L;
    /* synthetic */static Class class$demo$LineChartDemo1;
    public LineChartDemo1(String string) {
        super(string);
        JPanel jpanel = createDemoPanel();
        jpanel.setPreferredSize(new Dimension(500, 270));
        setContentPane(jpanel);
    }
    private static CategoryDataset createDataset() {
        DefaultCategoryDataset defaultcategorydataset = new Defa
ultCategoryDataset();
        defaultcategorydataset.addValue(212.0, "Classes", "JDK 1
.0");
        defaultcategorydataset.addValue(504.0, "Classes", "JDK 1
.1");
        defaultcategorydataset.addValue(1520.0, "Classes", "JDK
1.2");
        defaultcategorydataset.addValue(1842.0, "Classes", "JDK
1.3");
        defaultcategorydataset.addValue(2991.0, "Classes", "JDK
1.4");
        defaultcategorydataset.addValue(3500.0, "Classes", "JDK
1.5");
        return defaultcategorydataset;
    }
    private static JFreeChart createChart(CategoryDataset catego
rydataset) {
        JFreeChart jfreechart = ChartFactory.createLineChart(
            "Java Standard Class Library",// 图表标题
            null, // 主轴标签
            "Class Count",// 范围轴标签
            categorydataset, // 数据集
            PlotOrientation.VERTICAL,// 方向
            false, // 是否包含图例
            true, // 提示信息是否显示
            false// 是否使用urls
        );
        // 添加主标题
        jfreechart.addSubtitle(new TextTitle("Number of Classes
By Release"));
        TextTitle texttitle = (new TextTitle(
            "Source: Java In A Nutshell (5th Edition) by David F
lanagan (O'Reilly)"));
        texttitle.setFont(new Font("SansSerif", 0, 10));
        texttitle.setPosition(RectangleEdge.BOTTOM);
        texttitle.setHorizontalAlignment(HorizontalAlignment.RIG
HT);
        jfreechart.addSubtitle(texttitle);
        // 改变图表的背景颜色
```

```
        jfreechart.setBackgroundPaint(Color.white);
        CategoryPlot categoryplot = (CategoryPlot) jfreechart.ge
tPlot();
        categoryplot.setBackgroundPaint(Color.lightGray);
        categoryplot.setRangeGridlinePaint(Color.white);
        categoryplot.setRangeGridlinesVisible(false);
        URL url = (class$demo$LineChartDemo1 == null ? class$dem
o$LineChartDemo1 = class$("demo.LineChartDemo1")
            : class$demo$LineChartDemo1).getClassLoader().getRes
ource(
            "OnBridge11small.png");
        if (url != null) {
            ImageIcon imageicon = new ImageIcon(url);
            jfreechart.setBackgroundImage(imageicon.getImage());
            categoryplot.setBackgroundPaint(new Color(0, 0, 0, 0
));
        }
        NumberAxis numberaxis = (NumberAxis) categoryplot.getRan
geAxis();
        numberaxis.setStandardTickUnits(NumberAxis.createInteger
TickUnits());
        LineAndShapeRenderer lineandshaperenderer = (LineAndShap
eRenderer) categoryplot
            .getRenderer();
        lineandshaperenderer.setShapesVisible(true);
        lineandshaperenderer.setDrawOutlines(true);
        lineandshaperenderer.setUseFillPaint(true);
        lineandshaperenderer.setBaseFillPaint(Color.white);
        lineandshaperenderer.setSeriesStroke(0, new BasicStroke(
3.0F));
        lineandshaperenderer.setSeriesOutlineStroke(0, new Basic
Stroke(2.0F));
        lineandshaperenderer.setSeriesShape(0, new Ellipse2D.Dou
ble(-5.0, -5.0,
            10.0, 10.0));
        LineAndShapeRenderer renderer = (LineAndShapeRenderer) c
ategoryplot
            .getRenderer();
        renderer.setShapesVisible(true);
        renderer.setDrawOutlines(true);
        renderer.setUseFillPaint(true);
        return jfreechart;
    }
    public static JPanel createDemoPanel() {
        JFreeChart jfreechart = createChart(createDataset());
        return new ChartPanel(jfreechart);
    }
    public static void main(String[] strings) {
        LineChartDemo1 linechartdemo1 = new LineChartDemo1(
        "JFreeChart - Line Chart Demo 1");
        linechartdemo1.pack();
        RefineryUtilities.centerFrameOnScreen(linechartdemo1);
        linechartdemo1.setVisible(true);
```

```
    }
    /* synthetic */
    static Class class$(String string) {
        Class var_class;
        try {
            var_class = Class.forName(string);
        } catch (ClassNotFoundException classnotfoundexception)
{
            throw new NoClassDefFoundError(classnotfoundexceptio
n.getMessage());
        }
        return var_class;
    }
}
```

7.3 使用XYDataset数据集创建折线图

7.3.1 概述

折线图也可以使用XYDataset数据集，使用一条直线将相邻的点（x，y）点连接起来。本章介绍的一个使用XYDataset数据集创建折线图的简单实例，如下图7.2所示。

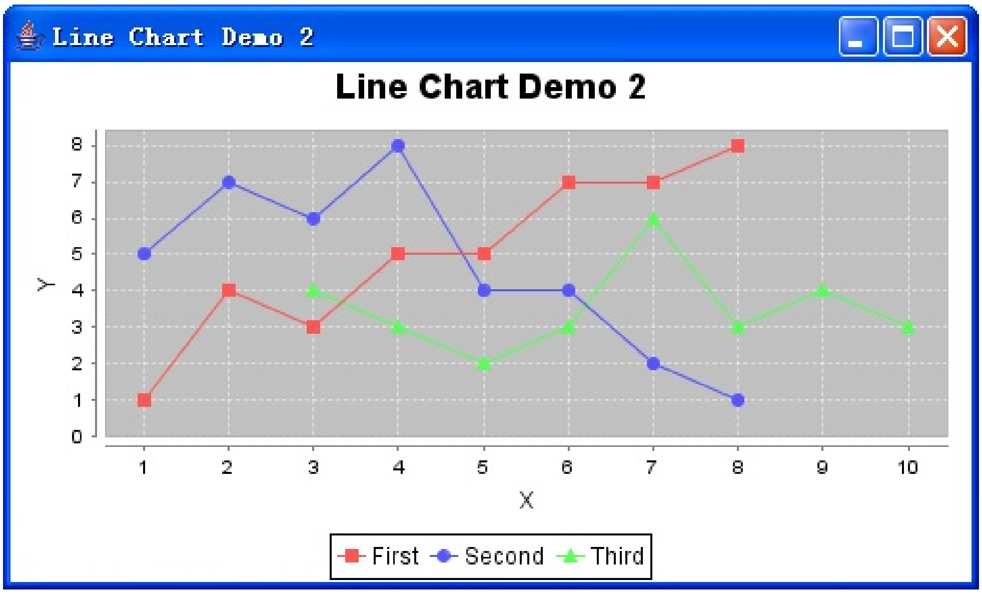


图7.2 一个简单的基于XYDataset数据集的折线图（参考：LineChartDemo2.java）

7.3.2 XYDataset

对于该图表来说，使用的数据集是XYSeriesCollection（当然我们可以使用实现XYDataset接口的其他数据集）。出于独立演示的目点，我们创建的dataset代码如下：

```
private static XYDataset createDataset() {
    XYSeries xyseries = new XYSeries("First");
    xyseries.add(1.0, 1.0);
    xyseries.add(2.0, 4.0);
    xyseries.add(3.0, 3.0);
    xyseries.add(4.0, 5.0);
    xyseries.add(5.0, 5.0);
    xyseries.add(6.0, 7.0);
    xyseries.add(7.0, 7.0);
    xyseries.add(8.0, 8.0);
    XYSeries xyseries_0_ = new XYSeries("Second");
    xyseries_0_.add(1.0, 5.0);
    xyseries_0_.add(2.0, 7.0);
    xyseries_0_.add(3.0, 6.0);
    xyseries_0_.add(4.0, 8.0);
    xyseries_0_.add(5.0, 4.0);
    xyseries_0_.add(6.0, 4.0);
    xyseries_0_.add(7.0, 2.0);
    xyseries_0_.add(8.0, 1.0);
    XYSeries xyseries_1_ = new XYSeries("Third");
    xyseries_1_.add(3.0, 4.0);
    xyseries_1_.add(4.0, 3.0);
    xyseries_1_.add(5.0, 2.0);
    xyseries_1_.add(6.0, 3.0);
    xyseries_1_.add(7.0, 6.0);
    xyseries_1_.add(8.0, 3.0);
    xyseries_1_.add(9.0, 4.0);
    xyseries_1_.add(10.0, 3.0);
    XYSeriesCollection xyseriescollection = new XYSeriesCollecti
on();
    xyseriescollection.addSeries(xyseries);
    xyseriescollection.addSeries(xyseries_0_);
    xyseriescollection.addSeries(xyseries_1_);
    return xyseriescollection;
}
```

注意：每个系列必须有x值（不是必须有y值），并且该系列独立于其他系列。数据集可以接受一个y值为null的值。当图表遇到null值时，连接线不被画出，该系列的连线不会连续。出现下图7.3类型。

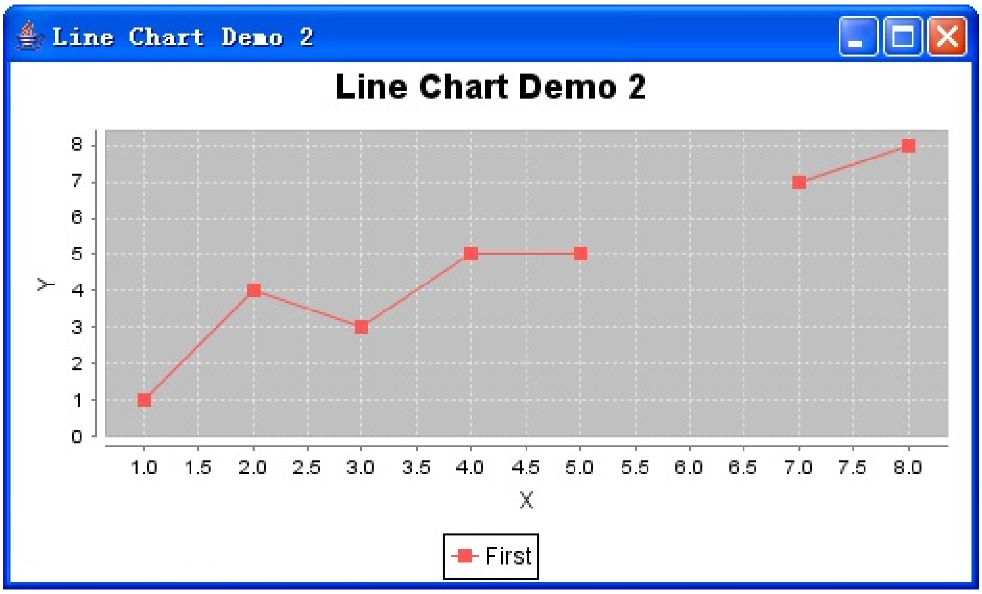


图7.3有一个 y值为null时，图表显示断续。

7.3.3 创建图表

ChartFactory类提供了一个便利的方法createXYLineChart()创图表：

```
JFreeChart jfreechart = ChartFactory.createXYLineChart(
    "Line Chart Demo 2", // chart title
    "X", // x axis label
    "Y", // y axis label
    xydataset, // data
    PlotOrientation.VERTICAL,
    true, // include legend
    true, // tooltips
    false // urls
);
```

上面方法构建了一个JFreeChart对象，该对象具有一个标题、图例和相关轴的图区及renderer。数据集使用上节我们创建的数据集。

7.3.4 定制图表

图表将使用大部分缺省的属性进行初始化设置。当然了，我们也可以随意修改这些属性，来改变我们图表的外观。在本实例中，设置的几个属性如下：

- 设置图表的背景颜色
- 设置图区的背景颜色
- 设置轴的平移值
- 设置主轴和范围轴网格线颜色
- 修改renderer改变连线点的形状
- 范围轴刻度的设置，以便显示整数值。

改变图表背景颜色非常简单。代码如下：

```
//改变图表的背景颜色
jfreechart.setBackgroundPaint(Color.white);
```

改变图区背景颜色、轴平移、网格线颜色，需要使用plot图区对象的一个引用来修改。图片对象需要转化成XYPlot对象，主要是因为我们可以访问更多更具体的图区方法。代码如下：

```
XYPlot xyplot = (XYPlot) jfreechart.getPlot();
xyplot.setBackgroundPaint(Color.lightGray);
xyplot.setAxisOffset(new RectangleInsets(5.0, 5.0, 5.0, 5.0));
xyplot.setDomainGridlinePaint(Color.white);
xyplot.setRangeGridlinePaint(Color.white);
```

修改renderer来显示连线之间的形状。代码如下：

```
XYLineAndShapeRenderer xylineandshaperenderer = (XYLineAndShapeR
enderer) xyplot.getRenderer();
xylineandshaperenderer.setShapesVisible(true);
xylineandshaperenderer.setShapesFilled(true);
```

最后就是修改范围轴。我们将默认刻度值（允许显示小数）改成只显示整数的刻度值。代码如下：

```
NumberAxis numberaxis = (NumberAxis) xyplot.getRangeAxis();
numberaxis.setStandardTickUnits(NumberAxis.createIntegerTickUnit
s());
```

参考源代码、Javadoc的API文档以及其他相关XYPlot的定制内容，来学习更多的细节。

8 时序图

8.1 简介

时序图类似于折线图，唯一不同的地方是时序图的主轴是日期而不是数值。本章讲述如何使用JFreeChart创建时序图。

8.2 创建时序图

8.2.1 概述

时序图表的确是一个使用XYDataset数据集的折线图。不同点就是再主轴上X轴值显示的是日期。本章讲述的一个简单应用如下图8.1所示

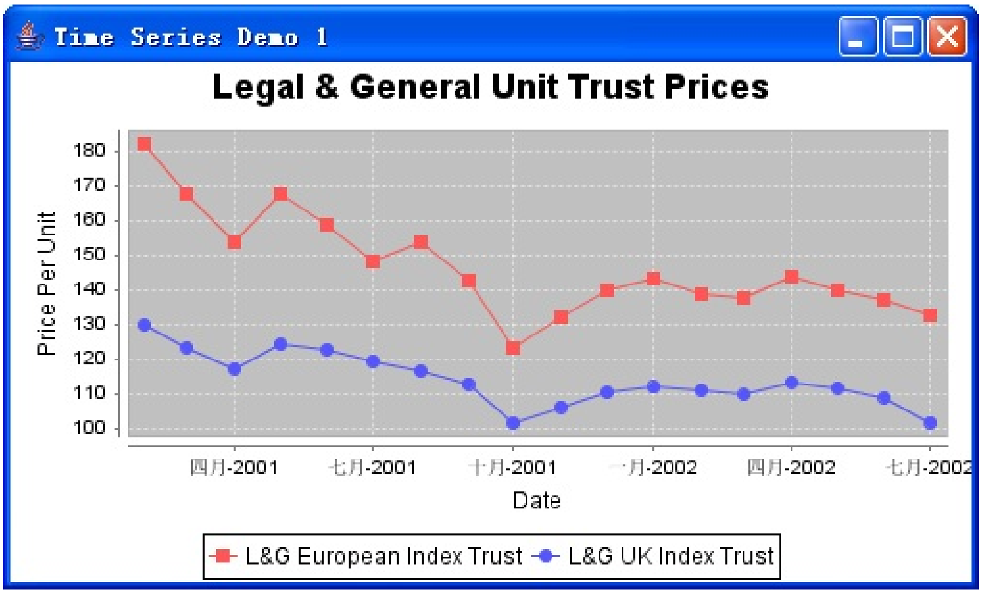


图8.1 一个简单的时序图表（参考：TimeSeriesDemo1.java）。

上图实例代码参见类TimeSeriesDemo1.java。

8.2.2 日期还是数字？

创建序图使用的数据集是XYDataset。接口不能有返回日期类型以外的方法。那么JFreeChart是如何创建时序图表的呢？

数据集返回的x值是基本的double类型，但这个值通过一种特殊的方式来进行转译成日期——该数是反映了一个从1970/1/1起计算的一个毫秒级值（译码过程使用java.util.Data类计算）。

具体的轴类（DateAxis）将毫秒级数据转化成日期，并作为需要返回该值，将数值作为主轴刻度显示出来。

8.2.3 数据集

演示的本实例，数据使用的是一个TimeSeriesCollection对象（我们可以是任何XYDataset接口的实现）。代码如下：

```
private static XYDataset createDataset() {
    TimeSeries timeseries = new TimeSeries(
```

```
        "L&G European Index Trust",
        (class$org$jfree$data$time$Month == null ? (class$org$jf
ree$data$time$Month = class$("org.jfree.data.time.Month"))
        : class$org$jfree$data$time$Month));
    timeseries.add(new Month(2, 2001), 181.8);
    timeseries.add(new Month(3, 2001), 167.3);
    timeseries.add(new Month(4, 2001), 153.8);
    timeseries.add(new Month(5, 2001), 167.6);
    timeseries.add(new Month(6, 2001), 158.8);
    timeseries.add(new Month(7, 2001), 148.3);
    timeseries.add(new Month(8, 2001), 153.9);
    timeseries.add(new Month(9, 2001), 142.7);
    timeseries.add(new Month(10, 2001), 123.2);
    timeseries.add(new Month(11, 2001), 131.8);
    timeseries.add(new Month(12, 2001), 139.6);
    timeseries.add(new Month(1, 2002), 142.9);
    timeseries.add(new Month(2, 2002), 138.7);
    timeseries.add(new Month(3, 2002), 137.3);
    timeseries.add(new Month(4, 2002), 143.9);
    timeseries.add(new Month(5, 2002), 139.8);
    timeseries.add(new Month(6, 2002), 137.0);
    timeseries.add(new Month(7, 2002), 132.8);
    TimeSeries timeseries_0_ = new TimeSeries(
        "L&G UK Index Trust",
        (class$org$jfree$data$time$Month == null ? (class$org$jf
ree$data$time$Month = class$("org.jfree.data.time.Month"))
        : class$org$jfree$data$time$Month));
    timeseries_0_.add(new Month(2, 2001), 129.6);
    timeseries_0_.add(new Month(3, 2001), 123.2);
    timeseries_0_.add(new Month(4, 2001), 117.2);
    timeseries_0_.add(new Month(5, 2001), 124.1);
    timeseries_0_.add(new Month(6, 2001), 122.6);
    timeseries_0_.add(new Month(7, 2001), 119.2);
    timeseries_0_.add(new Month(8, 2001), 116.5);
    timeseries_0_.add(new Month(9, 2001), 112.7);
    timeseries_0_.add(new Month(10, 2001), 101.5);
    timeseries_0_.add(new Month(11, 2001), 106.1);
    timeseries_0_.add(new Month(12, 2001), 110.3);
    timeseries_0_.add(new Month(1, 2002), 111.7);
    timeseries_0_.add(new Month(2, 2002), 111.0);
    timeseries_0_.add(new Month(3, 2002), 109.6);
    timeseries_0_.add(new Month(4, 2002), 113.2);
    timeseries_0_.add(new Month(5, 2002), 111.6);
    timeseries_0_.add(new Month(6, 2002), 108.8);
    timeseries_0_.add(new Month(7, 2002), 101.6);
    TimeSeriesCollection timeseriescollection = new TimeSeriesCo
llection();
    timeseriescollection.addSeries(timeseries);
    timeseriescollection.addSeries(timeseries_0_);
    return timeseriescollection;
}
```

实例中，系列包含了每月的数据。尽管如此，仍然使用TimeSeries类来显示间隔的时间值（年、日、小时等）。

8.2.4 构建图表

使用ChartFactory类提供的便利方法createTimeSeriesChart()创建图表，代码如下：

```
JFreeChart jfreechart = ChartFactory.createTimeSeriesChart(
    "Legal & General Unit Trust Prices", // title
    "Date", // x-axis label
    "Price Per Unit", // y-axis label
    xydataset, // data
    true, // create legend?
    true, // generate tooltips?
    false // generate URLs?
);
```

该方法构建了一个带有标题、图例、相应轴的区域和展示器的JFreeChart对象。使用的数据集见上一节内容。

8.2.5 定制图表

图表的大部分属性使用了缺省的值进行初始化。当然，我们可以随时修改这些属性的设置来改变我们图表的外观展现。在本实例中，修改的几个属性如下：

- 修改renderer，改变每个数据点显示的系列形状，数据点之间的折线除外。
- 主轴的数据格式进行格式化后显示。

修改renderer需要一下两个步骤：获得renderer引用和将renderer对象转化成XYLineAndShapeRenderer类型。代码如下：

```
XYItemRenderer xyitemrenderer = xyplot.getRenderer();
if (xyitemrenderer instanceof XYLineAndShapeRenderer) {
    XYLineAndShapeRenderer xylineandshaperenderer = (XYLineAndSh
apeRenderer) xyitemrenderer;
    xylineandshaperenderer.setBaseShapesVisible(true);
    xylineandshaperenderer.setBaseShapesFilled(true);
}
```

最后，将格式化的数据传给主轴，以改变显示。代码如下：

```
DateAxis dateaxis = (DateAxis) xyplot.getDomainAxis();
dateaxis.setDateFormatOverride(new SimpleDateFormat("MMM-yyyy"))
;
```

当设置了Dataaxis时，系统将自动选择一个DataTickUnit来显示主轴刻度。但系统将使用上面我们格式化的数据来显示，而不是系统默认的格式。

8.2.6 全部代码

文档的全部代码如下：

```
/* TimeSeriesDemo1 - Decompiled by JODE
* Visit http://jode.sourceforge.net/
*/
package demo;
import java.awt.Color;
import java.awt.Dimension;
import java.text.SimpleDateFormat;
import javax.swing.JPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.time.Month;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
import org.jfree.data.xy.XYDataset;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RectangleInsets;
import org.jfree.ui.RefineryUtilities;
public class TimeSeriesDemo1 extends ApplicationFrame {
    private static final long serialVersionUID = -54122863709566
46368L;
    /* synthetic */
    static Class class$org$jfree$data$time$Month;
    public TimeSeriesDemo1(String string) {
        super(string);
        XYDataset xydataset = createDataset();
        JFreeChart jfreechart = createChart(xydataset);
        ChartPanel chartpanel = new ChartPanel(jfreechart, false
);
        chartpanel.setPreferredSize(new Dimension(500, 270));
        chartpanel.setMouseZoomable(true, false);
        setContentPane(chartpanel);
    }
    private static JFreeChart createChart(XYDataset xydataset) {
        JFreeChart jfreechart = ChartFactory.createTimeSeriesCha
rt(
            "Legal & General Unit Trust Prices", // title
            "Date", // x-axis label
            "Price Per Unit", // y-axis label
            xydataset, // data
            true, // create legend?
```

```
                true, // generate tooltips?
                false // generate URLs?
            );
            jfreechart.setBackgroundPaint(Color.white);
            XYPlot xyplot = (XYPlot) jfreechart.getPlot();
            xyplot.setBackgroundPaint(Color.lightGray);
            xyplot.setDomainGridlinePaint(Color.white);
            xyplot.setRangeGridlinePaint(Color.white);
            xyplot.setAxisOffset(new RectangleInsets(5.0, 5.0, 5.0,
5.0));
            xyplot.setDomainCrosshairVisible(true);
            xyplot.setRangeCrosshairVisible(true);
            org.jfree.chart.renderer.xy.XYItemRenderer xyitemrendere
r = xyplot
                .getRenderer();
            if (xyitemrenderer instanceof XYLineAndShapeRenderer) {
                XYLineAndShapeRenderer xylineandshaperenderer = (XYL
ineAndShapeRenderer) xyitemrenderer;
                xylineandshaperenderer.setBaseShapesVisible(true);
                xylineandshaperenderer.setBaseShapesFilled(true);
            }
            DateAxis dateaxis = (DateAxis) xyplot.getDomainAxis();
            dateaxis.setDateFormatOverride(new SimpleDateFormat("MMM
-yyyy"));
            return jfreechart;
        }
        private static XYDataset createDataset() {
            TimeSeries timeseries = new TimeSeries(
                "L&G European Index Trust",
                (class$org$jfree$data$time$Month == null ? (class$or
g$jfree$data$time$Month = class$("org.jfree.data.time.Month"))
                : class$org$jfree$data$time$Month));
            timeseries.add(new Month(2, 2001), 181.8);
            timeseries.add(new Month(3, 2001), 167.3);
            timeseries.add(new Month(4, 2001), 153.8);
            timeseries.add(new Month(5, 2001), 167.6);
            timeseries.add(new Month(6, 2001), 158.8);
            timeseries.add(new Month(7, 2001), 148.3);
            timeseries.add(new Month(8, 2001), 153.9);
            timeseries.add(new Month(9, 2001), 142.7);
            timeseries.add(new Month(10, 2001), 123.2);
            timeseries.add(new Month(11, 2001), 131.8);
            timeseries.add(new Month(12, 2001), 139.6);
            timeseries.add(new Month(1, 2002), 142.9);
            timeseries.add(new Month(2, 2002), 138.7);
            timeseries.add(new Month(3, 2002), 137.3);
            timeseries.add(new Month(4, 2002), 143.9);
            timeseries.add(new Month(5, 2002), 139.8);
            timeseries.add(new Month(6, 2002), 137.0);
            timeseries.add(new Month(7, 2002), 132.8);
            TimeSeries timeseries_0_ = new TimeSeries(
                "L&G UK Index Trust",
                (class$org$jfree$data$time$Month == null ? (class$or
```

```
g$jfree$data$time$Month = class$("org.jfree.data.time.Month"))
            : class$org$jfree$data$time$Month));
        timeseries_0_.add(new Month(2, 2001), 129.6);
        timeseries_0_.add(new Month(3, 2001), 123.2);
        timeseries_0_.add(new Month(4, 2001), 117.2);
        timeseries_0_.add(new Month(5, 2001), 124.1);
        timeseries_0_.add(new Month(6, 2001), 122.6);
        timeseries_0_.add(new Month(7, 2001), 119.2);
        timeseries_0_.add(new Month(8, 2001), 116.5);
        timeseries_0_.add(new Month(9, 2001), 112.7);
        timeseries_0_.add(new Month(10, 2001), 101.5);
        timeseries_0_.add(new Month(11, 2001), 106.1);
        timeseries_0_.add(new Month(12, 2001), 110.3);
        timeseries_0_.add(new Month(1, 2002), 111.7);
        timeseries_0_.add(new Month(2, 2002), 111.0);
        timeseries_0_.add(new Month(3, 2002), 109.6);
        timeseries_0_.add(new Month(4, 2002), 113.2);
        timeseries_0_.add(new Month(5, 2002), 111.6);
        timeseries_0_.add(new Month(6, 2002), 108.8);
        timeseries_0_.add(new Month(7, 2002), 101.6);
        TimeSeriesCollection timeseriescollection = new TimeSeri
esCollection();
        timeseriescollection.addSeries(timeseries);
        timeseriescollection.addSeries(timeseries_0_);
        return timeseriescollection;
    }
    public static JPanel createDemoPanel() {
        JFreeChart jfreechart = createChart(createDataset());
        return new ChartPanel(jfreechart);
    }
    public static void main$(String[] strings) {
        TimeSeriesDemo1 timeseriesdemo1 = new TimeSeriesDemo1(
            "Time Series Demo 1");
        timeseriesdemo1.pack();
        RefineryUtilities.centerFrameOnScreen(timeseriesdemo1);
        timeseriesdemo1.setVisible(true);
    }
    public static void main(String[] args) {
        main$(args);
    }
    /* synthetic */
    static Class class$(String string) {
        Class var_class;
        try {
            var_class = Class.forName(string);
        } catch (ClassNotFoundException classnotfoundexception)
{
            throw new NoClassDefFoundError(classnotfoundexceptio
n.getMessage());
        }
        return var_class;
    }
}
```

9 定制图表（Customising Charts）

9.1 简介

JFreeChart的设计的定制功能是非常灵活的。我们可以使用非常多的属性来设置我们图表的外观。本章将详细介绍一些图表通用的定制技术。

9.2 图表属性

9.2.1 概述

我们可以使用JFreeChart类方法从更高的层次来定制我们图表的外观。可控制的属性有：

- 图表的边框
- 图表的标题和副标题
- 图表的背景颜色和图片
- 使用绘制建议（Rendering Hints）画图表，该属性有是否反锯齿功能。

在下面的章节中将详细描述这些内容。

9.2.2 图表边框

JFreeChart可以在图表的外围画出一个边框。默认状态下，JFreeChart是不画出边框的，但我们可以使用方法setBorderVisible()来设置。边框的颜色和线条风格可使用方法setBorderPaint()和setBorderStroke()来控制。

注意：如果我们在一个ChartPanel里面显示我们的图表，那么我们可能更愿意使用Swing提供的边框。

9.2.3 图表标题

图表有一个标题，显示在图表的顶部、底部、左侧或右侧（同时，我们也可以添加副标题，见下章讲述）。标题使用一个TextTitle的实例对象。我们可以使用getTitle()方法来获得标题的引用。

```
TextTitle texttitle = jfreechart.getTitle();
```

修改标题文本（不修改字体和位置）的代码如下：

```
texttitle.setText("Pie Chart Demo");
```

题头放置在图表的顶部、底部、左侧或右侧的设置，使用标题本书属性设置来完成。下面代码显示的是将标题移植到图表的底部。

```
texttitle.setPosition(RectangleEdge.BOTTOM);
```

如果在我们图表上，我们不希望显示标题，则将标题设置为null即可。

9.2.4 副标题

图表可以拥有任何数量的副标题。添加副标题，需要先创建一个副标题对象（任何Title类的子类），然后将该对象加到图表上即可。代码如下：

```
TextTitle subtitle1 = new TextTitle("A Subtitle");
jfreechart.addSubtitle(subtitle1);
```

我们可以在图表上添加任何数量的副标题，但是紧急我们添加的副标题越多，图表画图的区域就越小。

修改一个已有的副标题，我们需要先获得副标题的一个引用。代码如下：

```
Title subtitle = jfreechart.getSubtitle(0);
```

在我们改变副标题属性之前，我们需要将Title的引用转换成我们需要的适当的子类类型。

我们可以使用getSubtitleCount()方法获得副标题的数量。

9.2.5 设置图表背景颜色

我们可以使用setBackgroundPaint()方法设置图表的背景颜色（注意，我们也可以设置我们图区的背景颜色，这与图表的背景颜色不同）。例如：

```
jfreechart.setBackgroundPaint(Color.blue);
```

我们可使用Paint接口的任何实现作为背景颜色的设置参数，其中有Color、GradientPaint（渐变颜色）和TexturePaint等。代码如下：

```
Paint p = new GradientPaint(0, 0, Color.white, 1000, 0, Color.gr
een);
jfreechart.setBackgroundPaint(p);
```

我们可以设置我们的背景颜色为null，这时推荐使用一个背景图片来设置我们的图表。

9.2.6 使用背景图片

我们可以使用方法setBackgroundImage()来为我们的图表设置一幅背景图表。

```
jfreechart.setBackgroundImage(JFreeChart.INFO.getLogo());
```

默认的，图片充满图表的整个背景，图片失真。但我们可以使用setBackgroundImageAlignment()方法来改变图片不充满整个背景。代码如下：

```
jfreechart.setBackgroundImageAlignment(Align.TOP_LEFT);
```

使用setBackgroundImageAlpha()方法，我们可以控制图片的透明度。如果我们希望图片只填充我们图表的区域（区域包含轴），那么我们需要将背景图片添加到图表的图区。代码如下（以饼图为例）：

```
PiePlot pieplot = (PiePlot) jfreechart.getPlot();
pieplot.setBackgroundImage(JFreeChart.INFO.getLogo())
```

9.2.7 Rendering Hints（绘制建议）

JFreeChart使用java2D的API来画图表。在java2D中的API中，我们可以提供绘制建议让绘制引擎绘制图表。JFreeChart允许我们在画图表时，使用setRenderingHints()方法，将绘制建议参数传入java2D的API中。

JFreeChart还提供了一个便利反锯齿开关方法。当反锯齿开关开时，图表会绘制出比较光滑的图表，但是花费的时间要长。代码如下：

```
jfreechart.setAntiAlias(true);
```

JFreeChart画图时，默认为反锯齿开关为开。

9.3 图区属性

9.3.1 概述

JFreeChart类在绘制图表时，将大部分工作交给了Plot类（图形绘制结构）或Plot的子类。JFreeChart类的getPlot()方法返回了一个图表创建的图区（plot）的引用。

```
Plot plot = jfreechart.getPlot();
```

我们需要将该引用转化成Plot的一个具体子类。例如：

```
CategoryPlot plot = jfreechart.getCategoryPlot();
```

或

```
XYPlot plot = jfreechart.getXYPlot();
```

注意：如果plot不是相应的类，则在转化的时候，会抛出ClassCastException类型转制异常。

9.3.2 图区子类

那么我们如何知道我们图表使用的Plot是那个子类呢？作为使用JFreeChart的经验，分清那些图表使用CategoryPlot和那些图表使用XYPlot是非常清晰的。如果还怀疑，看一下ChartFactory类的源代码就会明白每个类型的图表是如何放在一起的。

9.3.3 设置图区背景颜色

我们可以使用方法setBackgroundPaint()设置图区的背景颜色。例如：

```
Plot plot = jfreechart.getPlot();
plot.setBackgroundPaint(Color.white);
```

我们可使用Paint接口的任何实现作为背景颜色的设置参数，其中有Color、GradientPaint（渐变颜色）和TexturePaint等。同时，我们也可以设置背景颜色为null。

9.3.4 设置背景图片

我们可以使用方法setBackgroundImage()为图区设置备有图片。

```
Plot plot = jfreechart.getPlot();
plot.setBackgroundImage(JFreeChart.INFO.getLogo());
```

默认的，图片充满图表的整个背景，图片失真。但我们可以改变图片不充满整个背景，使用方法是setBackgroundImageAlignment()。

```
plot.setBackgroundImageAlignment(Align.BOTTOM_RIGHT);
```

使用setBackgroundImageAlpha()方法，我们可以控制图片的透明度。如果我们希望图片充满这个图表区域，那么我们需要将背景图片添加到JFreeChart对象上（前面已经介绍过）。

9.4 轴属性

9.4.1 概述

使用JFreeChart创建的大部分图表都带有两个轴。X轴和Y轴。当然对于一些图表（比如饼图）根本就没有轴。对于使用轴的图表来说，图区使用Axis对象来管理轴。

9.4.2 获得轴对象引用

在你修改轴的属性之前，我们需要先获得一个轴的引用。图区类CategoryPlot和XYPlot类有两个方法getDomainAxis()和getRangeAxis()分别是获得X轴Y轴对象。这两个方法返回了一个ValueAxis对象的引用，除了在使用CategoryPlot的情况下，X轴使用的是CategoryAxis。代码如下：

```
// get an axis reference...
CategoryPlot plot = jfreechart.getCategoryPlot();
CategoryAxis domainAxis = plot.getDomainAxis();
// change axis properties...
domainAxis.setLabel("Categories");
domainAxis.setLabelFont(someFont);
```

CategoryAxis和ValueAxis类有许多不同的子类。有时我们需要将轴对象引用转化成具体的子类，为了获取更多具体的属性。如，如果我们想获得y轴为一个对象NumberAxis。代码如下：

```
XYPlot plot = jfreechart.getXYPlot();
NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
rangeAxis.setAutoRange(false);
```

9.4.3 设置轴标签

我们使用方法setLabel()可以改变轴的标签。如果我们不想在图表的轴上有标签，那么我们就设置为null即可。

我们可以使用Axis类定义的方法setLabelFont(), setLabelPaint(), 和setLabelInsets()改变标签的字体、颜色等内容。

9.4.4 改变周边标签显示方向

当图区在左侧或右侧画一个轴（水平轴）时，轴标签会自动旋转90度，以满足小空间的需要。如果我们希望标签也水平，我们需要修改标签的角度：

```
XYPlot plot = jfreechart.getXYPlot();
ValueAxis axis = plot.getRangeAxis();
axis.setLabelAngle(Math.PI / 2.0);
```

注意角度的表示使用弧度（PI为180度）。

9.4.5 隐藏刻度标签

隐藏某个轴的刻度标签：

```
CategoryPlot plot = jfreechart.getCategoryPlot();
ValueAxis axis = plot.getRangeAxis();
axis.setTickLabelsVisible(false);
```

对于CategoryAxis，方法setTickLabelsVisible(false)隐藏种类标签。

9.4.6 隐藏刻度符号

隐藏某个轴的刻度符号：

```
XYPlot plot = jfreechart.getXYPlot();
Axis axis = plot.getDomainAxis();
axis.setTickMarksVisible(false);
```

注意category轴没有刻度符号。

9.4.7 设置刻度尺寸

默认的，数值和日期会自动选择一个刻度尺寸，以便刻度标签不会重复显示。但我们也可以使用setTickUnit()方法设置我们自己的饿刻度单位。

9.4.8 指定标准的数值刻度单位

在NumberAxis类中，方法允许我们设置我们自己的刻度单位替代系统自动选择刻度danwi的机制。最普通的应用就是我们有一个仅仅显示整数的数轴。在实例中，我们不想让0.5或者0.25作为刻度单位。在NumberAxis类中有一个静态方法返回一系列的标准整数刻度单位：

```
XYPlot plot = jfreechart.getXYPlot();
NumberAxis axis = (NumberAxis) plot.getRangeAxis();
TickUnitSource units = NumberAxis.createIntegerTickUnits();
axis.setStandardTickUnits(units);
```

如果我们想控制标准的刻度单位时，我们可以自由定制自己的TickUnits集合。

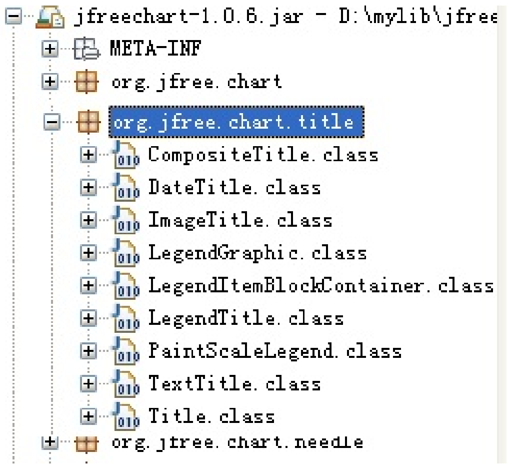
9.4.9 指定标准的日期刻度单位

类似于上一节内容，DateAxis类也有一个setStandardTickUnits()方法，来设置我们的刻度单位。方法createStandardDateTickUnits()为DateAxis返回了一个缺省的集合。同时我们也可以创建我们自己的标准日期刻度单位。

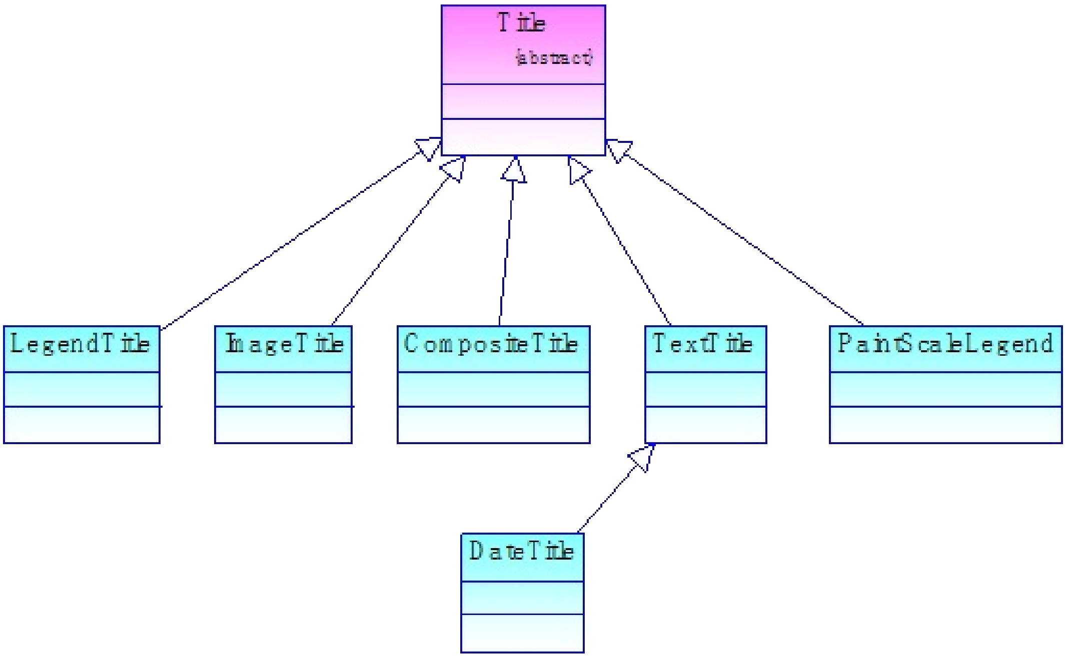
9.5 心得体会

9.5.1 Title子类如下图：

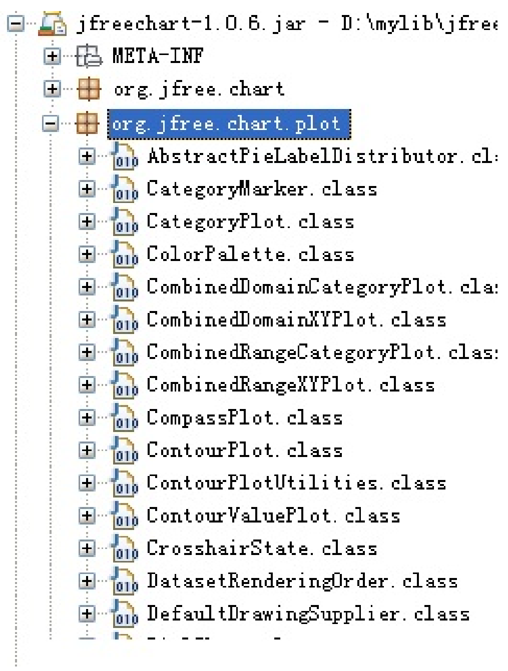
包图如下所示：



关系类图如下：

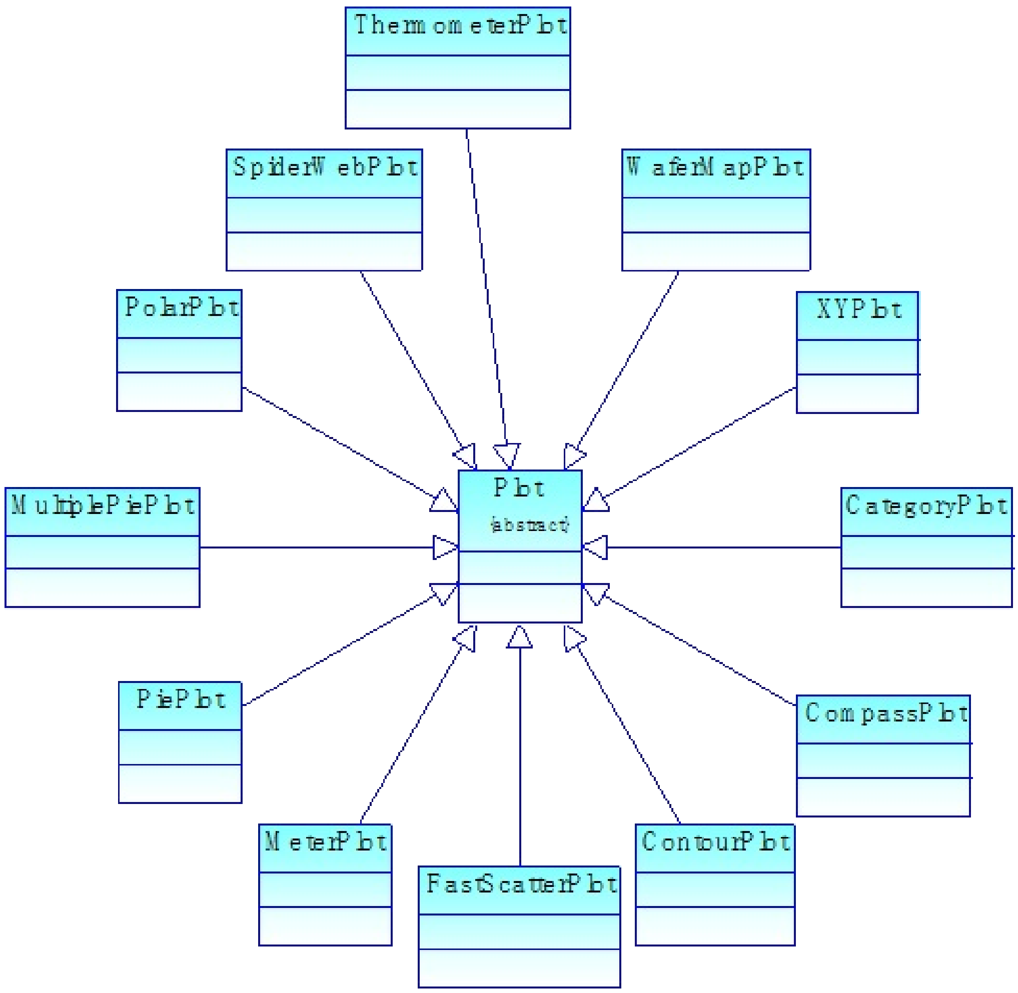


9.5.2 Plot类



jfreechart-1.0.6.jar - D:\mylib\jfree
META-INF
org.jfree.chart
org.jfree.chart.plot
AbstractPieLabelDistributor.cl
CategoryMarker.class
CategoryPlot.class
ColorPalette.class
CombinedDomainCategoryPlot.cla
CombinedDomainXYPlot.class
CombinedRangeCategoryPlot.clas
CombinedRangeXYPlot.class
CompassPlot.class
ContourPlot.class
ContourPlotUtilities.class
ContourValuePlot.class
CrosshairState.class
DatasetRenderingOrder.class
DefaultDrawingSupplier.class

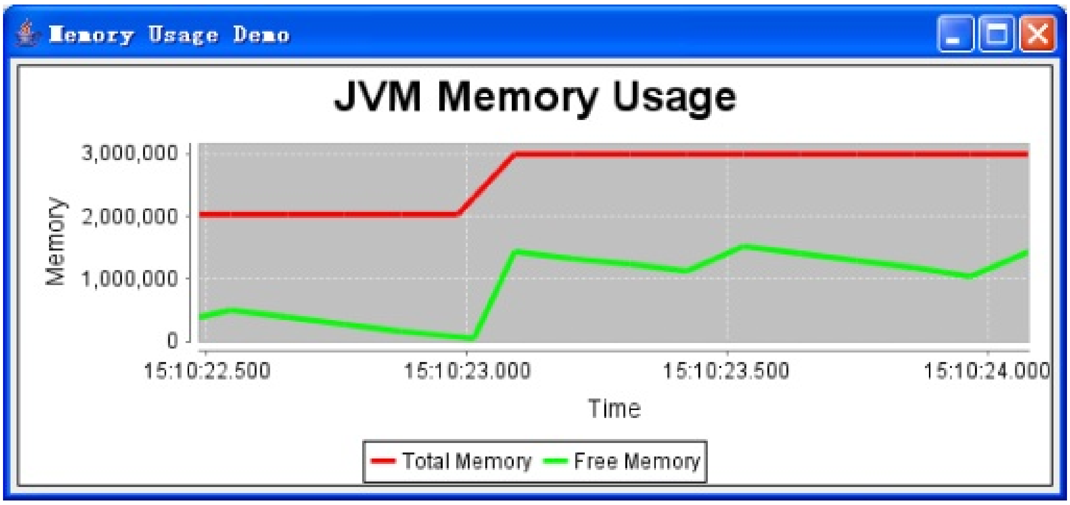
类结构图如下：



10 动态图（Dynamic Charts）

10.1 简介

为了说明使用JFreeChart创建“动态”的图表，本章节阐述了一个简单的应用说明这个过程，该应用为动态刷新JVM内存显示已使用和未使用情况。如下图10.1



如图10.1 一个简单的“动态”实例（参考：MemoryUsageDemo.java）。

10.2 知识背景

10.2.1 事件监听

JFreeChart使用监听机制来响应其他chart组件改变的响应。例如，不论数据源在何时发生更新，一个DatasetChangeEvent事件总被发送给已注册进数据源的监听器。

响应触发发生一系列事件：

- 图区监听到数据源改变的通知。如果需要更新轴的值，然后将PlotChangeEvent事件通知所有注册的监听器。
- 图表监听到图区更改事件的通知，然后将ChartChangeEvent事件通知给所有注册的监听器。
- 最后，ChartPanel接受到该面板上显示的图表的更改事件，ChartPanel根据响应的事件画出响应的图表——完全重新画，而不是仅仅更新数据。

所有图表或者其他子控件改变发生的事件过程都遵循上面的过程。

10.2.2 性能优化

关于性能优化，我们必须明白JFreeChart不会产生实时图表。每次数据源的更新，ChartPanel都需要重新画全部的图表。

性能优化通常是非常困难的。 比如，JFreeChart调用图像2D的API提取最新变更的点，从而只画更新的点。我们使用JFreeChart完成这个实时过程的实例将限制了“每秒产生页面”的数量。产生数量的大小是否是一个瓶颈的关键问题，主要取决我们我们画图表所依赖的数据，应用的环境和操作环境。

10.3 实例应用

10.3.1 概述

实例MemoryUsageDemo.java文档可以从下面的链接中获得：

http://www.object-refinery.com/jfreechart/premium/index.html

页面需要输入购买JFreeChart开发指南时需要的用户名和密码。

10.3.2 创建一个dataset

创建的实例代码数据源，包含了一个单一的时序集合，集合内有两个TimeSeries对象（一个是计算总内存，另一个是计算剩余内存）。代码如下：

```
this.total = new TimeSeries("Total", Millisecond.class);
this.total.setMaximumItemAge(30000);
this.free = new TimeSeries("Free", Millisecond.class);
this.free.setMaximumItemAge(30000);
TimeSeriesCollection dataset = new TimeSeriesCollection();
dataset.addSeries(this.total);
dataset.addSeries(this.free);
```

每个时间系列的maximumItemAge属性设置为30000毫秒（30秒）。因此任何时候添加到新系列的新数据，都是30秒前记录的老数据。

10.3.3 创建一个图表

图表的创建（定制）都遵循所有图表创建的标准模式。创建动态图也是一样，没有任何特殊的步骤。除了我们将autoRange属性设置为true之外。同时，这也有利于维护图表使用的数据源的引用。

10.3.4 更新一个dataset

在本实例演示中，通过向两个时序图添加的数据来更新数据源，该数据的添加有一个独立的线程Timer管理。代码如下：

```
class DataGenerator extends Timer implements ActionListener {
    DataGenerator(int i) {
        super(i, null);
        addActionListener(this);
    }
    public void actionPerformed(ActionEvent actionevent) {
        long l = Runtime.getRuntime().freeMemory();
        long l_0_ = Runtime.getRuntime().totalMemory();
        MemoryUsageDemo.this.addTotalObservation((double) l_0_);
        MemoryUsageDemo.this.addFreeObservation((double) l);
    }
}
```

注意JFreeChart在画图表和数据源更新代码之间没有使用线程同步，因此是不安全的。另一点需要注意的是，曾做过一个关于JFreeChart内存泄露问题的测试，将JFreeChart的实例在一个机器上连续运行6天。随着图表的不断更新，我们就可以看到垃圾收集器所产生的影响。六天之后，发现总内存的使用量保持不变。当JFreChart产生并丢弃的临时对象（垃圾对象）时，剩余可用内存减少了。增加的使用内存量是垃圾收集器工作时所使用的。

10.3.5 全部代码

下面是全部的实例代码：

```
/* MemoryUsageDemo - Decompiled by JODE
* Visit http://jode.sourceforge.net/
*/
package demo;
import java.awt.BasicStroke;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.Timer;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.time.Millisecond;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
```

```
import org.jfree.ui.RectangleInsets;
public class MemoryUsageDemo extends JPanel {
    private static final long serialVersionUID = 677671283835949
8649L;
    private TimeSeries total;
    private TimeSeries free;
    /* synthetic */
    static Class class$org$jfree$data$time$Millisecond;
    class DataGenerator extends Timer implements ActionListener
{
        private static final long serialVersionUID = 1L;
        DataGenerator(int i) {
            super(i, null);
            addActionListener(this);
        }
        public void actionPerformed(ActionEvent actionevent) {
            long l = Runtime.getRuntime().freeMemory();
            long l_0_ = Runtime.getRuntime().totalMemory();
            MemoryUsageDemo.this.addTotalObservation((double) l_
0_);
            MemoryUsageDemo.this.addFreeObservation((double) l);
        }
    }
    public MemoryUsageDemo(int i) {
        super(new BorderLayout());
        total = new TimeSeries(
            "Total Memory",
            (class$org$jfree$data$time$Millisecond == null ? (cl
ass$org$jfree$data$time$Millisecond = class$("org.jfree.data.tim
e.Millisecond"))
            : class$org$jfree$data$time$Millisecond));
        total.setMaximumItemAge((long) i);
        free = new TimeSeries(
            "Free Memory",
            (class$org$jfree$data$time$Millisecond == null ? (cl
ass$org$jfree$data$time$Millisecond = class$("org.jfree.data.tim
e.Millisecond"))
            : class$org$jfree$data$time$Millisecond));
        free.setMaximumItemAge((long) i);
        TimeSeriesCollection timeseriescollection = new TimeSeri
esCollection();
        timeseriescollection.addSeries(total);
        timeseriescollection.addSeries(free);
        DateAxis dateaxis = new DateAxis("Time");
        NumberAxis numberaxis = new NumberAxis("Memory");
        dateaxis.setTickLabelFont(new Font("SansSerif", 0, 12));
        numberaxis.setTickLabelFont(new Font("SansSerif", 0, 12)
);
        dateaxis.setLabelFont(new Font("SansSerif", 0, 14));
        numberaxis.setLabelFont(new Font("SansSerif", 0, 14));
        XYLineAndShapeRenderer xylineandshaperenderer = new XYLi
neAndShapeRenderer(
            true, false);
```

```
        xylineandshaperenderer.setSeriesPaint(0, Color.red);
        xylineandshaperenderer.setSeriesPaint(1, Color.green);
        xylineandshaperenderer.setSeriesStroke(0, new BasicStrok
e(3.0F, 0, 2));
        xylineandshaperenderer.setSeriesStroke(1, new BasicStrok
e(3.0F, 0, 2));
        XYPlot xyplot = new XYPlot(timeseriescollection, dateaxi
s, numberaxis,
            xylineandshaperenderer);
        xyplot.setBackgroundPaint(Color.lightGray);
        xyplot.setDomainGridlinePaint(Color.white);
        xyplot.setRangeGridlinePaint(Color.white);
        xyplot.setAxisOffset(new RectangleInsets(5.0, 5.0, 5.0,
5.0));
        dateaxis.setAutoRange(true);
        dateaxis.setLowerMargin(0.0);
        dateaxis.setUpperMargin(0.0);
        dateaxis.setTickLabelsVisible(true);
        numberaxis.setStandardTickUnits(NumberAxis.createInteger
TickUnits());
        JFreeChart jfreechart = new JFreeChart("JVM Memory Usage
", new Font(
            "SansSerif", 1, 24), xyplot, true);
        jfreechart.setBackgroundPaint(Color.white);
        ChartPanel chartpanel = new ChartPanel(jfreechart, true)
;
        chartpanel.setBorder(BorderFactory.createCompoundBorder(
BorderFactory
            .createEmptyBorder(4, 4, 4, 4), BorderFactory
            .createLineBorder(Color.black)));
        add(chartpanel);
    }
    private void addTotalObservation(double d) {
        total.add(new Millisecond(), d);
    }
    private void addFreeObservation(double d) {
        free.add(new Millisecond(), d);
    }
    public static void main(String[] strings) {
        JFrame jframe = new JFrame("Memory Usage Demo");
        MemoryUsageDemo memoryusagedemo = new MemoryUsageDemo(30
000);
        jframe.getContentPane().add(memoryusagedemo, "Center");
        jframe.setBounds(200, 120, 600, 280);
        jframe.setVisible(true);
        memoryusagedemo.new DataGenerator(100).start();
        jframe.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent windowevent) {
                System.exit(0);
            }
        });
    }
    /* synthetic */static Class class$(String string) {
```

```
        Class var_class;
        try {
            var_class = Class.forName(string);
        } catch (ClassNotFoundException classnotfoundexception)
{
            throw new NoClassDefFoundError(classnotfoundexceptio
n.getMessage());
        }
        return var_class;
    }
}
```

11 图表工具条（Tooltips）

11.1 概述

JFreeChart为图表的每个组件提供了一套产生、收集和显示工具条的机制。本章主要介绍：

- 如何产生图表工具条（包括定制图表工具条）
- 如何收集图表工具条
- 如何显示图表工具条
- 如何隐藏图表工具条

11.2 创建图表工具条

如果我们需要使用图表工具条，我们首先确保所画的图表中已经产生图表工具条。我们可以为我们的图区或图区条目设置图表工具条产生器。在下面的相关章节里面，我们将了解如何为一个图表设置一个图表工具条。

11.2.1 饼图

饼图类PiePlot使用PieToolTipGenerator接口产生接口图表工具条。系统通过了该接口的一个标准实现类StandardPieToolTipGenerator。PiePlot设置图表工具条的方法如下：

```
public void setToolTipGenerator(PieToolTipGenerator generator);
```

该方法可以为饼图设置工具条产生器，如果设置null，则表示没有工具条。

11.2.2 generated.种类图

种类图表—包括JFreeChart创建最多的直方条形图—基于CategoryPlot类并使用CategoryItemRenderer来画每一个数据条目。Renderer使用接口CategoryToolTipGenerator的指定方法来获得图表工具条。为种类图区条目设置图表工具条产生器，使用类AbstractCategoryItemRenderer的方法：

```
public void setToolTipGenerator(CategoryToolTipGenerator generat
or);
```

该方法可以为饼图设置工具条产生器，如果设置null，则表示没有工具条。

11.2.3 XY图

XY图表—包括JFreeChart创建的散点图和时序图—基于类XYPlot并使用XYItemRenderer画出每一个数据条目。Renderer使用一个XYToolTipGenerator产生图表工具条。

设置XY图区条目的工具条，使用在AbstractXYItemRenderer定义的方法：

```
public void setToolTipGenerator(XYToolTipGenerator generator);
```

如果设置产生器为null，表示没有图表工具条产生器。

11.3 收集图表工具条

使用ChartRenderingInfo类可收集图表工具条信息，以及图表的其他信息。我们首先要向JFreeChart的draw()方法传入该类实例，否则图表工具条信息将不被记录（即便是产生器已经被注册到图区的数据条目中）。

幸运的是，ChartPanel会自动的处理图表工具条的收集。因此如果我们使用ChartPanel显示我们的图表，就不用担心图表工具条的收集—因为ChartPanel已经为我们收集了。

11.4 显示图表工具条

使用ChartPanel类创建我们的图表时，图表工具条会自动显示出来。并且你可以为图区（或者图区的renderer）设置一个图表工具条。

我们可是使用类的方法设置显示或隐藏图表工具条。方法如下：

```
public void setDisplayToolTips(boolean flag);
```

11.5 隐藏图表工具条

最有效的方式就是将图表工具条设置为null。确保没有任何图表工具条信息产生，这样可以节省内存同时提供处理速度（特别是对于大数据源时，非常有好处）。

我们可是使用上节讲的方法使用ChartPanele类设置图表工具条的隐藏。

11.6 定制图表工具条

我们可以通过相应的图表工具条产生器接口对每一个图表工具条进行文本的各种操作。

12 图表条目标签（Item Label）

12.1 简介

12.1.1 概述

对于大多数的图表类型来说，JFreeChart允许我们在图表的每个条目上、或者内部、或者附近显示条目标签。例如，下图12.1 在每个条形图上显示出了真实的值。

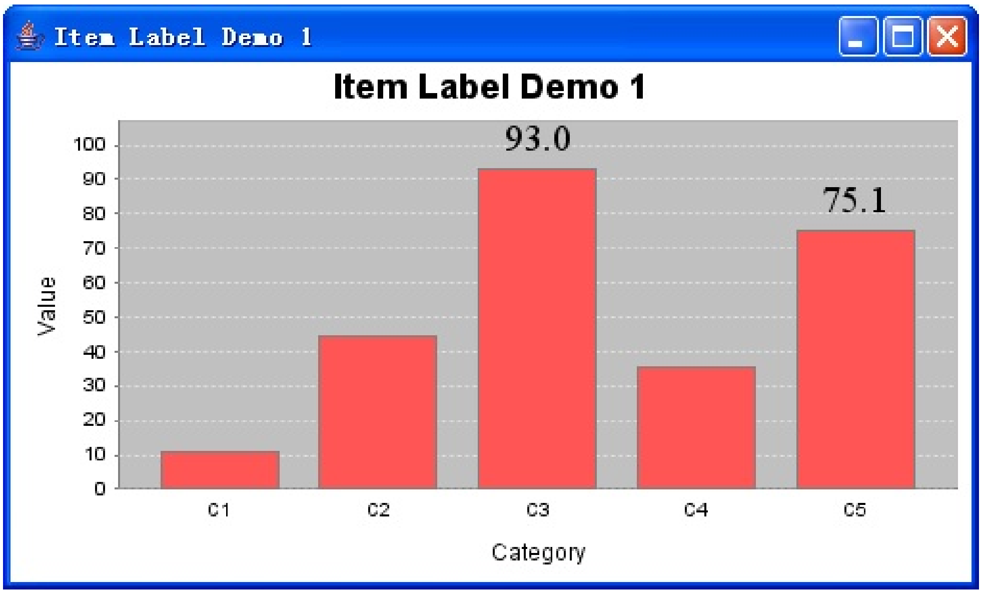


图12.1 显示数组的条形图（参考：）

本章主要讲述：

- 如何让条目标签可视（仅限于支持条目标签的图表类型）
- 如何改变条目标签的外观（字体和颜色）
- 如何指定条目标签的位置
- 如何定制条目标签的文本

忠告：我们使用上面的特征时，要谨慎。图表是期望用来分析总结数据的——如果我们觉得在图表上显示真实数据是非常有必要的话，那我们的数据应使用一个表格格式显示更为合适。

12.1.2 局限性

在当前版本JFreeChart中，条目标签的使用是有很多局限性的：

- 一些renderer不支持条目标签
- 轴范围的自动调节，忽略了条目标签的自动调整——如果图表的周围没有足够的空间（使用方法setUpperMargin()或setLowerMargin()进行了相应的调整），那么一些图表条目标签在图表上显示不出来。

相信，在以后的JFreeChart版本中，这些限制问题将被解决。

12.2 显示条目标签

12.2.1 概述

条目标签默认是不显示的，因此我们需要使用renderer进行创建和显示条目标签。这主要有以下两个步骤：

- 分配一个CategoryItemLabelGenerator或XYItemLabelGenerator给renderer—这是一个负责创建标签的对象。
- 在renderer里面设置一个标签可视的标志。可以针对全部系列进行设置，也可以针对具体的每一个系列进行设置。

此外，我们可以定制条目标签的位置、字体和颜色。在下面的章节里我们将详细的介绍。

12.2.2 创建一个条目标签并赋值

使用renderer分配的一个标签产生器创建条目标签（这与图表工具条的机制是相同的）。

下面代码说了将一个标签产生器指派给CategoryItemRenderer：

```
CategoryItemRenderer renderer = categoryplot.getRenderer();
CategoryItemLabelGenerator generator = new StandardCategoryItemL
abelGenerator("{2}", new DecimalFormat("0.00"));
renderer.setBaseItemLabelGenerator(generator);
```

同样的，将一个产生器指派给XYItemRenderer，代码如下：

```
XYPlot plot = (XYPlot) jfreechart.getPlot();
XYItemRenderer renderer = plot.getRenderer();
XYItemLabelGenerator generator = new StandardXYItemLabelGenerato
r(
    "{2}", new DecimalFormat("0.00"), new DecimalFormat("0.00"))
;
renderer.setBaseItemLabelGenerator(generator);
```

我们可以在标准产生器的构造函数中定制不同的行为。当然了，我们也可以创建我们总计的产生器，详见12.5.2章节。

12.2.3 所有的系列显示条目标签

方法renderer.setBaseItemLabelsVisible(false)是控制着条目标签的显示。对于CategoryItemRenderer：

```
CategoryItemRenderer renderer = categoryplot.getRenderer();
renderer.setBaseItemLabelsVisible(true);
```

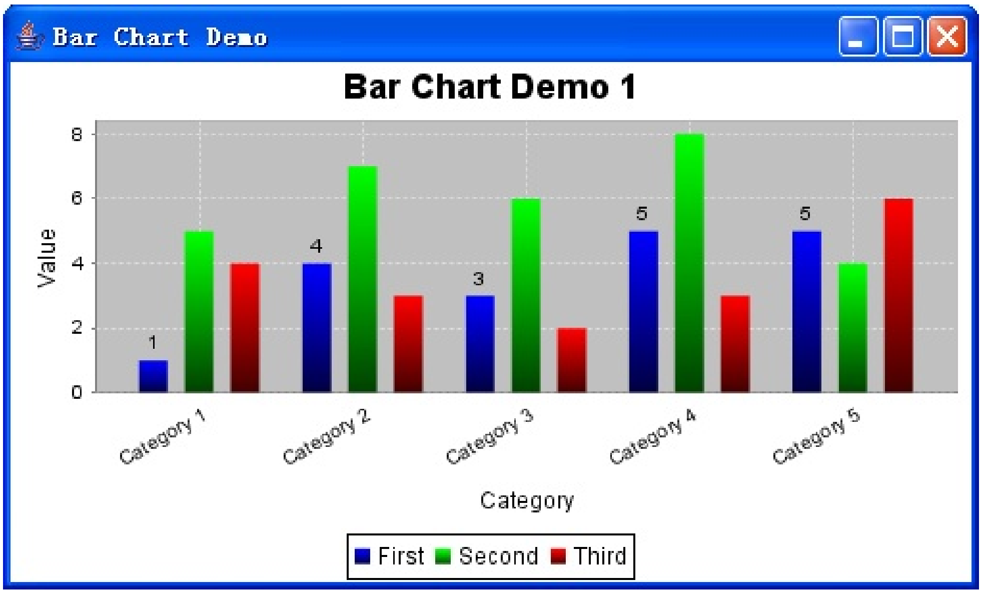
同样对于：XYItemRenderer

```
XYItemRenderer renderer = categoryplot.getRenderer();
renderer.setBaseItemLabelsVisible(true);
```

一旦设置，这个标志优先管理我们在所有地方对每一系列做的设置，主要为了应用每一系列的设置。我们可以设置个标志为null（见12.2.4章节）

12.2.4 为选择的系列显示条目标签

我们可以控制图表的每一个系列的条目标签是否显示。例如：如下图12.2仅显示第一系列条目标签。



如图12.2显示第一系列条目标签

下面代码可以设置如上效果：

```
CategoryItemRenderer renderer = categoryplot.getRenderer();
renderer.setBaseItemLabelGenerator(new StandardCategoryItemLabel
Generator());
renderer.setBaseItemLabelsVisible(null); // clears the ALL serie
s flag
renderer.setSeriesItemLabelsVisible(0, true);
renderer.setSeriesItemLabelsVisible(1, false);
```

注意：上面代码中对全部的系列设置为null—这一点非常重要，因为全部系列的标志控制每一个系列的标志。

12.2.5 问题与解决

如果按照上面的步骤操作，你仍然未看见条目标签显示在图表上，那么我们从以下几个方面进行考虑：

- Renderere必须需要一个标签产生器——这是一个用来创建每一个标签的文本条目的对象。
- 一些renderer不支持条目标签（具体参考renderer相关的文档）

12.3 条目标签外观

12.3.1 概述

我们可以通过改变条目的颜色、字体来改变图表条目标签的外观。正如其他renderer属性一样，属性的设置可以是全部的系列，可以是具体某一系列。

在JFreeChart目前的版本中，标签是月年个一个透明的背景画出来的。我们不能设置标签的背景颜色，也不能指定标签的边框。这些在以后的版本中会得到解决。

12.3.2 改变条目标签的字体

为了在所有的系列中改变条目标签的字体，我们可以使用下面的代码：

```
CategoryItemRenderer renderer = categoryplot.getRenderer();
renderer.setBaseItemLabelFont(new Font("黑体", Font.PLAIN, 20));
```

同样，也可以为单个系列设置字体：

```
// add settings for individual series...
renderer.setSeriesItemLabelFont(0, new Font("SansSerif", Font.PL
AIN, 20));
renderer.setSeriesItemLabelFont(1, new Font("SansSerif", Font.PL
AIN, 10));
```

注意：renderer.setBaseItemLabelFont(null)方法会出错。开发指南显示的代码有错误。

12.3.3 改变条目标签的颜色

改变条目标签的颜色，我们可以使用下面的代码：

```
CategoryItemRenderer renderer = categoryplot.getRenderer();
renderer.setBaseItemLabelPaint(Color.red);
```

同样的，可以为单独每一系列设置颜色：

```
// add settings for individual series...
renderer.setSeriesItemLabelPaint(0, Color.red);
renderer.setSeriesItemLabelPaint(1, Color.blue);
```

注意：renderer.setBaseItemLabelPaint(null);方法会出错。开发指南显示的代码有错误。

12.4 条目标签位置

12.4.1 概述

条目标签的位置是通过ItemLabelPosition对象的四个属性来控制的。

我们可以通过接口CategoryItemRenderer的方法来独立定义条目标签的正负点位置：

```
public void setBasePositiveItemLabelPosition(ItemLabelPosition p
osition);
public void setBaseNegativeItemLabelPosition(ItemLabelPosition p
osition);
```

理解这些属性如何影响独立标签的最终位置的关键是了解JFreeChart里面条目标签的特征。四个特征是：

- 条目标签点——决定标签的起始位置
- 文本点——标签里的文本相对于条目标签的位置。
- 旋转点——标签文本旋转的点位置
- 旋转角度——标签的旋转角度。

这些的详细描述在下一章详细介绍。

12.4.2 条目标签的位置

设置条目标签位置的目的，主要是为了找出标签在图表上贴向数据条目的一个点（x,y）位置。同时在画图表时，该标签也被画在该点处。更多的信息可以参考ItemLabelAnchor文档。

12.4.3 标签文本的位置

标签文本的位置，主要取决于上节讲的标签位置。我们可以讲标签文本在标签里设置在右上部、或左下部等，更多的信息参见TextAnchor文档。

运行JCommon包内的org.demo.package下面的DrawStringDemo应用，可以更好的理解标签文本在标签内是如何放置的。

12.4.4 标签旋转点

在标签上定义了一个旋转点，用于旋转标签。在DrawStringDemo实例中很好演示了这个特征。

12.4.5 标签旋转角度

旋转角度定义了标签沿旋转点旋转的角度。该角度为弧度。

12.5 定制条目标签文本

12.5.1 概述

定制条目标签文本，我们需要依赖用JFreeChart里的标签产生器来为条目标签创建文本。如果要想完全控制标签文本的控制，我们就需要编写自己的标签产生器，需要实现接口CategoryItemLabelGenerator。

在本章节里，我们对自定义标签器技术做了简要的讲述，然后用两个实例来说明该技术过程。

12.5.2 实现一个自定义的标签产生器

开发一个自定义标签产生器，我们需要写一个类，该类必须实现CategoryItemLabelGenerator接口里的方法。

```
public String generateLabel(CategoryDataset dataset, int series,
int category)
```

该renderer调用该方法获得一个标签的字符串，并且将该字符串传入到当前条目的CategoryDataset、序列和种类。这就意味着创建这个标签时，我们拥有完全的访问权限。

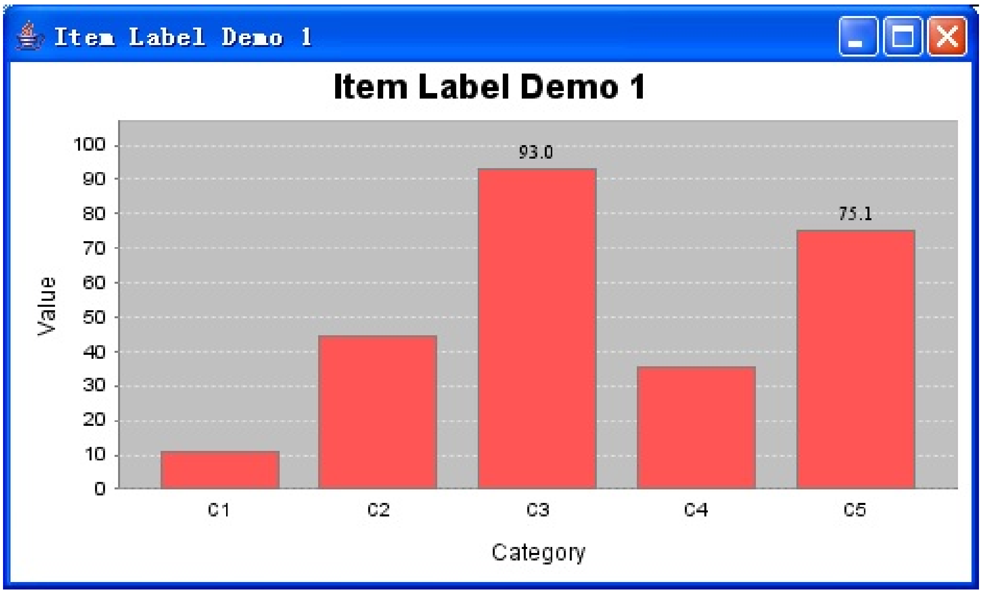
该方法可以返回任意字符串，因此我们格式化这个字符串。如果我们不想显示标签，可以设置为null。

在下面的两个例子中很好的说明了这个特征。

12.6 实例1

12.6.1 概述

在第一个实例中，目的就是当当条目的值大于某个限定的值时，就显示该标签。如图12.3所示。



如图12.3 超过某个限定值显示条目标签的实例

做到这一点并不困难，需要做以下工作：

- 写一个实现接口CategoryItemLabelGenerator的类，并且实现generateItemLabel()方法。该方法实现如果条目的值小于限定值时，返回null。
- 创建该类的实例，将该实例使用renderer的方法setLabelGenerator()设置到renderer中去。

12.6.2 源代码

```
package demo;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import javax.swing.JPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.labels.AbstractCategoryItemLabelGenerator
```

```
;
import org.jfree.chart.labels.CategoryItemLabelGenerator;
import org.jfree.chart.labels.StandardCategoryItemLabelGenerator
;
import org.jfree.chart.labels.StandardXYItemLabelGenerator;
import org.jfree.chart.labels.XYItemLabelGenerator;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.category.CategoryItemRenderer;
import org.jfree.chart.renderer.xy.XYItemRenderer;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;
public class ItemLabelDemo1 extends ApplicationFrame {
    static class LabelGenerator extends AbstractCategoryItemLabe
lGenerator
        implements CategoryItemLabelGenerator {
        private double threshold;
        public LabelGenerator(double d) {
            super("", NumberFormat.getInstance());
            threshold = d;
        }
        public String generateLabel(CategoryDataset categorydata
set, int i,
            int i_0_) {
            String string = null;
            Number number = categorydataset.getValue(i, i_0_);
            if (number != null) {
                double d = number.doubleValue();
                if (d &gt; threshold)
                string = number.toString();
            }
            return string;
        }
    }
    public ItemLabelDemo1(String string) {
        super(string);
        CategoryDataset categorydataset = createDataset();
        JFreeChart jfreechart = createChart(categorydataset);
        ChartPanel chartpanel = new ChartPanel(jfreechart);
        chartpanel.setPreferredSize(new Dimension(500, 270));
        setContentPane(chartpanel);
    }
    private static CategoryDataset createDataset() {
        DefaultCategoryDataset defaultcategorydataset = new Defa
ultCategoryDataset();
        defaultcategorydataset.addValue(11.0, "S1", "C1");
        defaultcategorydataset.addValue(44.3, "S1", "C2");
        defaultcategorydataset.addValue(93.0, "S1", "C3");
        defaultcategorydataset.addValue(35.6, "S1", "C4");
        defaultcategorydataset.addValue(75.1, "S1", "C5");
```

```
        return defaultcategorydataset;
    }
    private static JFreeChart createChart(CategoryDataset catego
rydataset) {
        JFreeChart jfreechart = ChartFactory.createBarChart(
            "Item Label Demo 1", "Category", "Value", categoryda
taset,
        PlotOrientation.VERTICAL, false, true, false);
        jfreechart.setBackgroundPaint(Color.white);
        CategoryPlot categoryplot = (CategoryPlot) jfreechart.ge
tPlot();
        categoryplot.setBackgroundPaint(Color.lightGray);
        categoryplot.setDomainGridlinePaint(Color.white);
        categoryplot.setRangeGridlinePaint(Color.white);
        NumberAxis numberaxis = (NumberAxis) categoryplot.getRan
geAxis();
        numberaxis.setUpperMargin(0.15);
        return jfreechart;
    }
    public static JPanel createDemoPanel() {
        JFreeChart jfreechart = createChart(createDataset());
        return new ChartPanel(jfreechart);
    }
    public static void main(String[] strings) {
        ItemLabelDemo1 itemlabeldemo1 = new ItemLabelDemo1("Item
 Label Demo 1");
        itemlabeldemo1.pack();
        RefineryUtilities.centerFrameOnScreen(itemlabeldemo1);
        itemlabeldemo1.setVisible(true);
    }
}
```

12.7 实例2

12.7.1 概述

在本实例中，目的是在每个系列的标签上显示出值和百分比值（这个百分比值，这个系列在某一部分的条形直方图或全部条形直方图的总值中的比值）。如下图12.4所示。

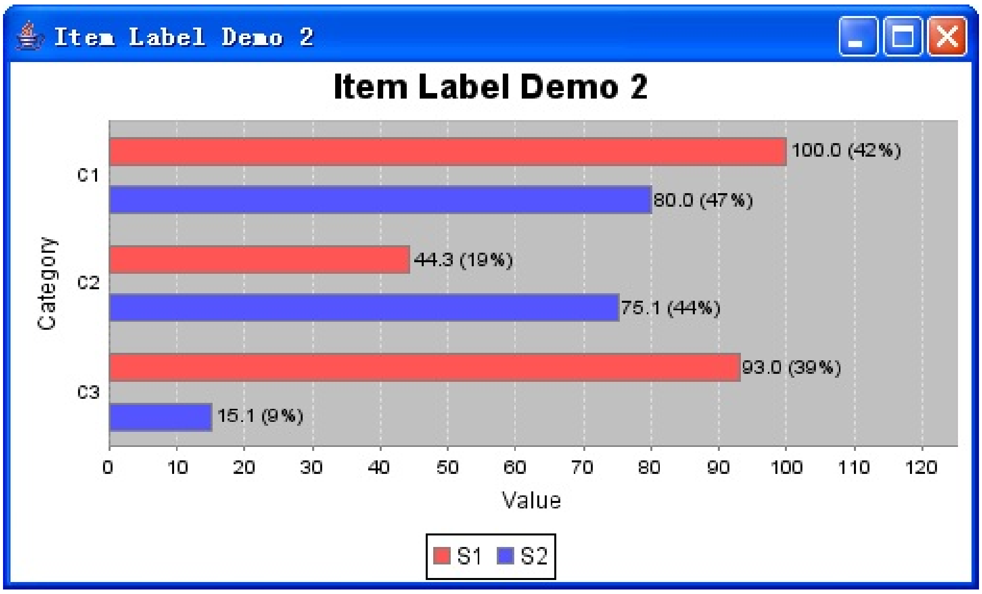


图12.4 带有比值的直方图

该实现中，标签产生器计算出百分比。如果传入构造函数的是一个种类索引，那么这个百分比的基数就是指定种类的当前系列的值。如果种类索引是无效的，那么这个基数就是指定种类的全部系列总和。

标签产生器会默认创建一个百分比格式——一种比较成熟的格式，提供格式化能力。

12.7.2 源代码

```
package demo;
import java.awt.Color;
import java.awt.Dimension;
import java.text.NumberFormat;
import javax.swing.JPanel;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.AxisLocation;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.labels.AbstractCategoryItemLabelGenerator
;
```

```
import org.jfree.chart.labels.CategoryItemLabelGenerator;
import org.jfree.chart.plot.CategoryPlot;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.renderer.category.CategoryItemRenderer;
import org.jfree.data.category.CategoryDataset;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.ui.ApplicationFrame;
import org.jfree.ui.RefineryUtilities;
public class ItemLabelDemo2 extends ApplicationFrame {
    static class LabelGenerator extends AbstractCategoryItemLabe
lGenerator
    implements CategoryItemLabelGenerator {
        private Integer category;
        private NumberFormat formatter = NumberFormat.getPercent
Instance();
        public LabelGenerator(int i) {
            this(new Integer(i));
        }
        public LabelGenerator(Integer integer) {
            super("", NumberFormat.getInstance());
            category = integer;
        }
        public String generateLabel(CategoryDataset categorydata
set, int i,
            int i_0_) {
            String string = null;
            double d = 0.0;
            if (category != null) {
                Number number = categorydataset
                .getValue(i, category.intValue());
                d = number.doubleValue();
            } else
                d = calculateSeriesTotal(categorydataset, i);
            Number number = categorydataset.getValue(i, i_0_);
            if (number != null) {
                double d_1_ = number.doubleValue();
                string = (number.toString() + " (" + formatter.f
ormat(d_1_ / d) + ")");
            }
            return string;
        }
        private double calculateSeriesTotal(CategoryDataset cate
gorydataset,
        int i) {
        double d = 0.0;
        for (int i_2_ = 0; i_2_ &lt; categorydataset.getColumnCo
unt(); i_2_++) {
        Number number = categorydataset.getValue(i, i_2_);
        if (number != null)
        d += number.doubleValue();
        }
        return d;
        }
```

```
    }
    public ItemLabelDemo2(String string) {
        super(string);
        CategoryDataset categorydataset = createDataset();
        JFreeChart jfreechart = createChart(categorydataset);
        ChartPanel chartpanel = new ChartPanel(jfreechart);
        chartpanel.setPreferredSize(new Dimension(500, 270));
        setContentPane(chartpanel);
    }
    private static CategoryDataset createDataset() {
        DefaultCategoryDataset defaultcategorydataset = new Defa
ultCategoryDataset();
        defaultcategorydataset.addValue(100.0, "S1", "C1");
        defaultcategorydataset.addValue(44.3, "S1", "C2");
        defaultcategorydataset.addValue(93.0, "S1", "C3");
        defaultcategorydataset.addValue(80.0, "S2", "C1");
        defaultcategorydataset.addValue(75.1, "S2", "C2");
        defaultcategorydataset.addValue(15.1, "S2", "C3");
        return defaultcategorydataset;
    }
    private static JFreeChart createChart(CategoryDataset catego
rydataset) {
        JFreeChart jfreechart = ChartFactory.createBarChart(
            "Item Label Demo 2", "Category", "Value", categoryda
taset,
        PlotOrientation.HORIZONTAL, true, true, false);
        jfreechart.setBackgroundPaint(Color.white);
        CategoryPlot categoryplot = (CategoryPlot) jfreechart.ge
tPlot();
        categoryplot.setBackgroundPaint(Color.lightGray);
        categoryplot.setDomainGridlinePaint(Color.white);
        categoryplot.setRangeGridlinePaint(Color.white);
        categoryplot.setRangeAxisLocation(AxisLocation.BOTTOM_OR
_LEFT);
        NumberAxis numberaxis = (NumberAxis) categoryplot.getRan
geAxis();
        numberaxis.setUpperMargin(0.25);
        CategoryItemRenderer categoryitemrenderer = categoryplot
.getRenderer();
        categoryitemrenderer.setBaseItemLabelsVisible(true);
        categoryitemrenderer.setBaseItemLabelGenerator(new Label
Generator(
            (Integer) null));
        return jfreechart;
    }
    public static JPanel createDemoPanel() {
        JFreeChart jfreechart = createChart(createDataset());
        return new ChartPanel(jfreechart);
    }
    public static void main(String[] strings) {
        ItemLabelDemo2 itemlabeldemo2 = new ItemLabelDemo2("Item
 Label Demo 2");
        itemlabeldemo2.pack();
```

```
            RefineryUtilities.centerFrameOnScreen(itemlabeldemo2);
            itemlabeldemo2.setVisible(true);
        }
    }
```

13 多轴和数据源图表（Multi Axis and Dataset）

13.1 简介

J FreeChart在CategoryPlot和XYPlot类中支持多轴和数据源显示。我们利用这个特征可以在一个图表上显示两个或多个数据源数据，但对于数据包含的数据有巨大差距时留有一定的余地。如图13.1所示。

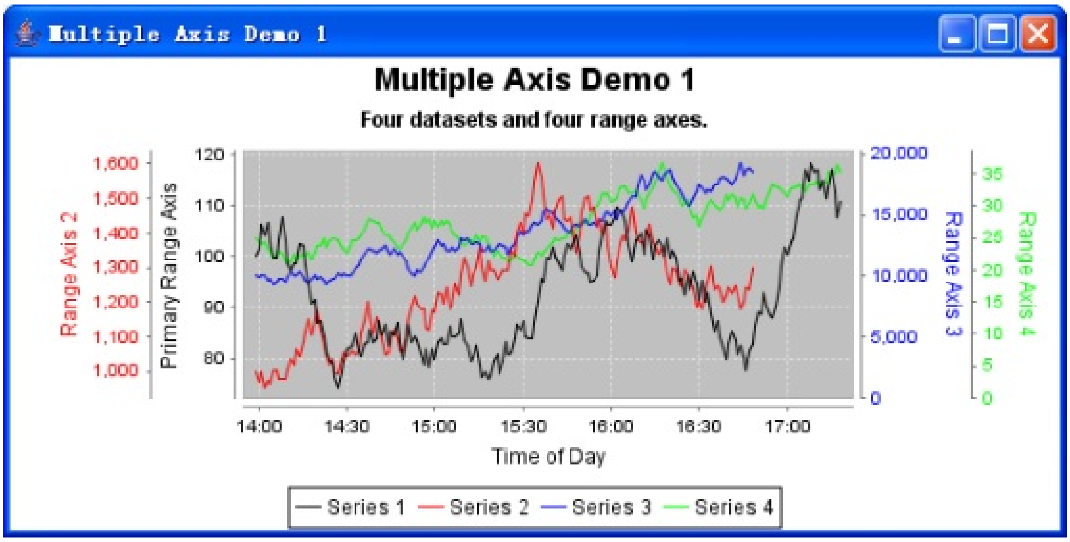


图13.1 具有多轴的图表

典型的，使用JFreeChart构建图表时，图表有一个单数据源、单renderer、单X/Y轴的图区最为常见。然而，在一个图区上添加多个数据源、多个renderer和多个轴也是可能的。在本章的实例中，展示了如何在一个图区上显示其他额外的数据源、renderer和轴。

13.2 实例

13.2.1 简介

MultipleAxisDemo1.java例子提供了一个很好的实例演示如何在一个图表上创建多轴的应用。本章在每一步的代码中提供了很多建议，详见后面的章节。

13.2.2 创建一个图表

创建一个具有多轴、多数据源、多renderer的图表，我们首先要创建一个常规的图表（例如使用ChartFactory类创建）。在本实例中，创建了一个时序图，代码如下：

```
XYDataset dataset1 = createDataset("Series 1", 100.0, new Minute
(), 200);
JFreeChart chart = ChartFactory.createTimeSeriesChart(
    "Multiple Axis Demo 1",
    "Time of Day",
    "Primary Range Axis",
    dataset1,
    true,
    true,
    false
);
```

13.2.3 添加额外的轴

如果在图区上添加额外的轴，我们使用setRangeAxis()方法来添加：

```
NumberAxis numberaxis = new NumberAxis("Range Axis 2"); xyplot.s
etRangeAxis(1, numberaxis);
xyplot.setRangeAxisLocation(1, AxisLocation.BOTTOM_OR_LEFT);
```

方法setRangeAxis()是用来添加图区的轴，注意轴的索引1已经被使用——我们添加其他轴时，通过增加该索引来添加新轴。方法setRangeAxisLocation()允许我们指定轴出现的位置（使用AxisLocation类）。我们添加的轴可以跟主坐标轴同一边，或者在对立边。例如：如果指定的是AxisLocation.BOTTOM_OR_LEFT，这意味着如果图区的方向是垂直的话，将在右边添加了一个Y轴，如果图区的方向是水平的话，将在底部添加一个Y轴。

在这里，图表上每一添加多余的数据源，因此如果我们显示该图表，我们将看到图上显示多轴，但轴上无数据显示。

13.2.4 添加一个额外的数据源

在图区上添加一个额外的数据源，使用setDataset()方法：

```
XYDataset xydataset_0_ = createDataset("Series 2", 1000.0,
new Minute(), 170);
xyplot.setDataset(1, xydataset_0_);
```

缺省的，数据源将使用主轴来显示数据。如果使数据源在另外的轴上显示数据，需使用方法mapDatasetToDomainAxis()和mapDatasetToRangeAxis()。这两个方法接受两个参数，第一个参数是数据源的索引，第二个是轴的索引。

13.2.5 添加一个额外的renderer

当我们添加一个数据源时，通常为该数据源添加一个附加的renderer也是非常有意义的。使用方法setRenderer()：

```
StandardXYItemRenderer standardxyitemrenderer = new StandardXYIt
emRenderer();
xyplot.setRenderer(1, standardxyitemrenderer);
```

方法的第一个参数为上节中添加的数据源的索引。注意：如果我们不想为数据源指定一个附加的rendere，系统将默认使用主renderer，这样系列的颜色就会在主数据源和附加数据源之间共享。

13.3 建议和技巧

当我们使用多轴图表时，我们需要为系列对应的轴提供一些可视化的建议。比如在例子MultipleAxisDemo1.java中轴标签的颜色与系列颜色是相匹配的。

可以从下面的实例中学习更多的技巧：

- DualAxisDemo1.java
- DualAxisDemo2.java
- DualAxisDemo3.java
- DualAxisDemo4.java
- MultipleAxisDemo1.java
- MultipleAxisDemo2.java
- MultipleAxisDemo3.java

14 组合图表（Combined Charts）

14.1 简介

JFreeChart支持几个图区类（可以管理着多个子类）组合而成的图表。图区类可以管理几个子类：

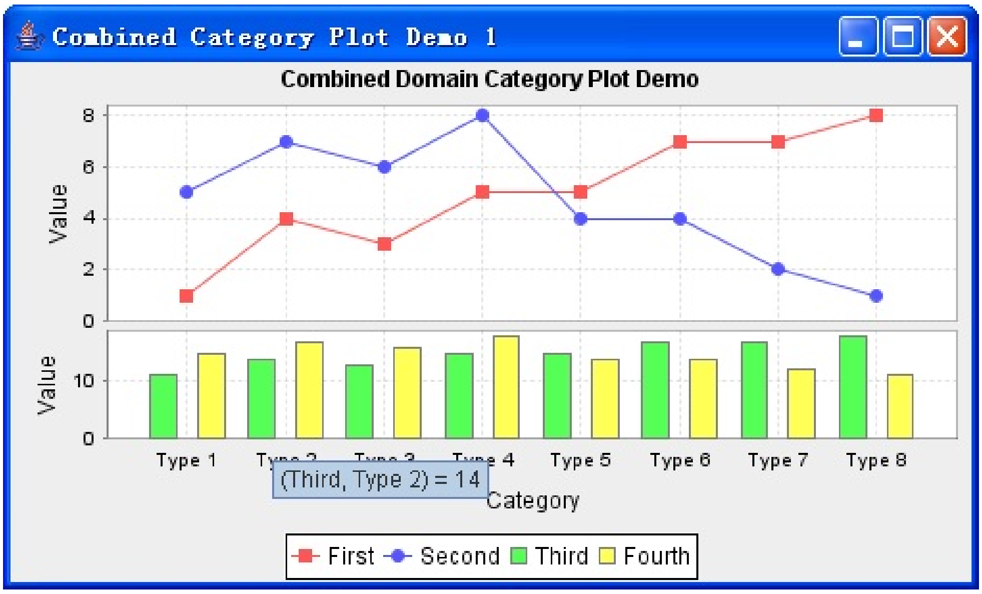
- `CombinedDomainCategoryPlot` / `CombinedRangeCategoryPlot`
- `CombinedDomainXYPlot` / `CombinedRangeXYPlot`；

本章使用几个实例说明了JFreeChart创建组合图表时的便利性。

14.2 组合X种类图区

14.2.1 概述

组合主域种类图区就是在一个图区上显示两个或者多个子图区（CategoryPlot实例），共享一个X轴的图区。每个子图区维护自己的Y轴。实例如图14.1所示。



如图14.1 组合X种类图区(共享X轴)

显示图表可以是水平的，也可以是垂直方向的——实例演示的是垂直的图表。

14.2.2 构建图表

提供了一个很好的例子，演示如何创建该图表的类型。关键的步骤是创建CombinedDomainCategoryPlot实例，然后添加两个子图区：

```
CategoryAxis domainAxis = new CategoryAxis("Category");
CombinedDomainCategoryPlot plot = new CombinedDomainCategoryPlot
(domainAxis);
plot.add(subplot1, 2);
plot.add(subplot2, 1);
JFreeChart result = new JFreeChart(
    "Combined Domain Category Plot Demo",
    new Font("SansSerif", Font.BOLD, 12),
    plot,
    true
);
```

注意，我们subplot1添加码值时是2（方法add()的第二个参数），而subplot1添加的是1呢？因为这控制着分配给各个图区的空间大小。

子图区的CategoryPlot实例对象将它们的X轴设置为null。例如在演示的实例中，代码如下：

```
CategoryDataset dataset1 = createDataset1();
NumberAxis rangeAxis1 = new NumberAxis("Value");
rangeAxis1.setStandardTickUnits(NumberAxis.createIntegerTickUnit
s());
LineAndShapeRenderer renderer1 = new LineAndShapeRenderer();
renderer1.setBaseToolTipGenerator(new StandardCategoryToolTipGen
erator());
CategoryPlot subplot1 = new CategoryPlot(dataset1, null, rangeAx
is1, renderer1);
subplot1.setDomainGridlinesVisible(true);
CategoryDataset dataset2 = createDataset2();
NumberAxis rangeAxis2 = new NumberAxis("Value");
rangeAxis2.setStandardTickUnits(NumberAxis.createIntegerTickUnit
s());
BarRenderer renderer2 = new BarRenderer();
renderer2.setBaseToolTipGenerator(new StandardCategoryToolTipGen
erator());
CategoryPlot subplot2 = new CategoryPlot(dataset2, null, rangeAx
is2, renderer2);
subplot2.setDomainGridlinesVisible(true);
```

14.3 组合Y种类图区

14.3.1 概述

一个组合Y种类图区就是一个图区显示两个或两个以上的子图区（CategoryPlot实例），共享Y轴。如果14.2.

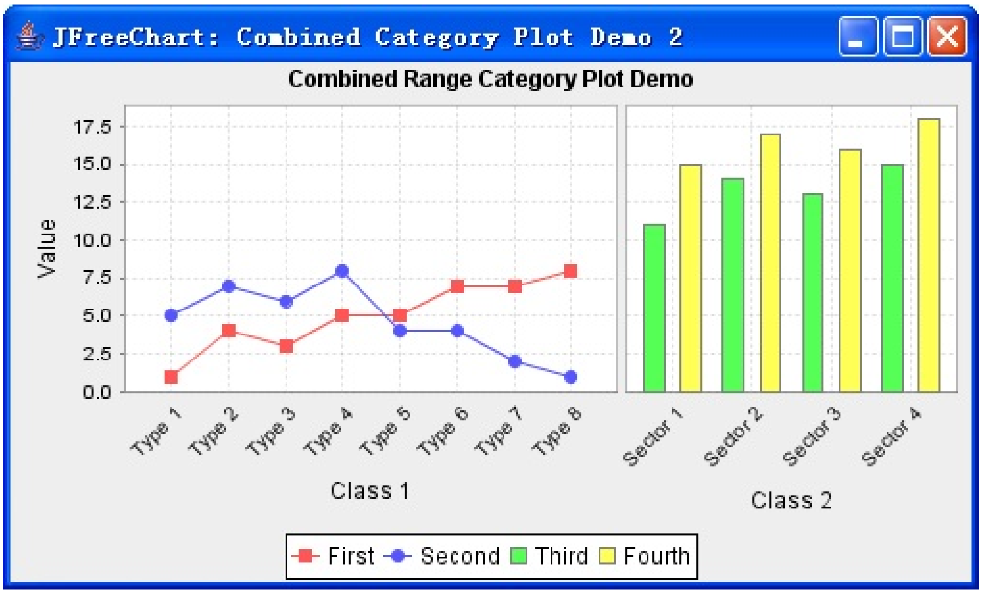


图14.2 组合Y种类图区

该图表可以水平显示也可以垂直显示（本例是垂直显示）。

14.3.2 构建图表

实例演示了如何创建该类型图表。关键的步骤是创建一个实例，然后添加两个子图区：

```
ValueAxis rangeAxis = new NumberAxis("Value");
```

CombinedRangeCategoryPlot plot = new CombinedRangeCategoryPlot(rangeAxis); plot.add(subplot1, 3); plot.add(subplot2, 2); JFreeChart result = new JFreeChart("Combined Range Category Plot Demo", new Font("SansSerif", Font.BOLD, 12), plot, true);

```
注意添加的子图区subplot1什么码值是3而子图区subplot2码值是2呢。这是因为该值控制这两个子图区分配的空间大小。

子图区是CategoryPlot实例，将Y轴设置为null。例如，在本实例演示的代码如下：
```

CategoryDataset dataset1 = createDataset1(); CategoryAxis domainAxis1 = new CategoryAxis("Class 1");
domainAxis1.setCategoryLabelPositions(CategoryLabelPositions.UP 45);
domainAxis1.setMaxCategoryLabelWidthRatio(5.0f); LineAndShapeRenderer renderer1 = new LineAndShapeRenderer();
renderer1.setBaseToolTipGenerator(new StandardCategoryToolTipGenerator());
CategoryPlot subplot1 = new CategoryPlot(dataset1, domainAxis1, null, renderer1); subplot1.setDomainGridlinesVisible(true); CategoryDataset dataset2 = createDataset2(); CategoryAxis domainAxis2 = new CategoryAxis("Class 2");
domainAxis2.setCategoryLabelPositions(CategoryLabelPositions.UP 45);
domainAxis2.setMaxCategoryLabelWidthRatio(5.0f); BarRenderer renderer2 = new BarRenderer(); renderer2.setBaseToolTipGenerator(new StandardCategoryToolTipGenerator()); CategoryPlot subplot2 = new CategoryPlot(dataset2, domainAxis2, null, renderer2);
subplot2.setDomainGridlinesVisible(true); ```

14.4 组合X-XY图区

14.4.1 概述

组合X-XY图区就是一个图区显示两个或者多个子图区（XYPlot实例），共享一个X轴。每一个子图区维护自己的Y轴。如下图14.3所示.

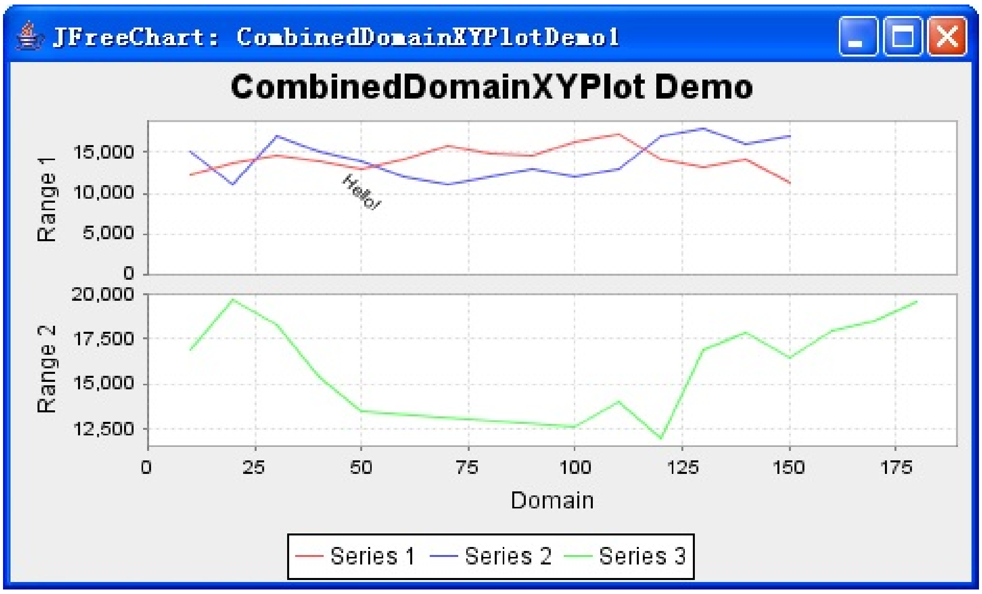


图14.3 组合X-XY图区（参见：CombinedXYPlotDemo5.java）

图区可能水平显示也可能垂直显示（本例子中垂直显示）。

14.4.2 构建图表

CombinedXYPlotDemo5.java实例演示了如何创建该类型的图表。关键的步骤是创建一个实例CombinedDomainXYPlot，并在该实例上添加两个子图区：

```
CombinedDomainXYPlot plot = new CombinedDomainXYPlot(new NumberA
xis("Domain"));
plot.setGap(10.0);
plot.add(subplot1, 1);
plot.add(subplot2, 1);
plot.setOrientation(PlotOrientation.VERTICAL);
return new JFreeChart(
    "CombinedDomainXYPlot Demo",
    JFreeChart.DEFAULT_TITLE_FONT, plot, true
);
```

注意两个图区的码值为什么都是1呢？因为该数值控制着每个图区分配的空间大小。

子图区是XYPlot实例，将自己的X轴设置为null。例如，下面的代码演示了这个特征：

```
XYDataset data1 = createDataset1();
XYItemRenderer renderer1 = new StandardXYItemRenderer();
NumberAxis rangeAxis1 = new NumberAxis("Range 1");
XYPlot subplot1 = new XYPlot(data1, null, rangeAxis1, renderer1)
;
subplot1.setRangeAxisLocation(AxisLocation.BOTTOM_OR_LEFT);
XYTextAnnotation annotation = new XYTextAnnotation("Hello!", 50.
0, 10000.0);
annotation.setFont(new Font("SansSerif", Font.PLAIN, 9));
annotation.setRotationAngle(Math.PI / 4.0);
subplot1.addAnnotation(annotation);
// create subplot 2...
XYDataset data2 = createDataset2();
XYItemRenderer renderer2 = new StandardXYItemRenderer();
NumberAxis rangeAxis2 = new NumberAxis("Range 2");
rangeAxis2.setAutoRangeIncludesZero(false);
XYPlot subplot2 = new XYPlot(data2, null, rangeAxis2, renderer2)
;
subplot2.setRangeAxisLocation(AxisLocation.TOP_OR_LEFT);
```

14.5 组合Y-XY图区

14.5.1 概述

组合Y-XY图区就是一个图区显示两个或者多个子图区（XYPlot实例），共享一个Y轴。每一个子图区维护自己的X轴。如下图14.4所示.

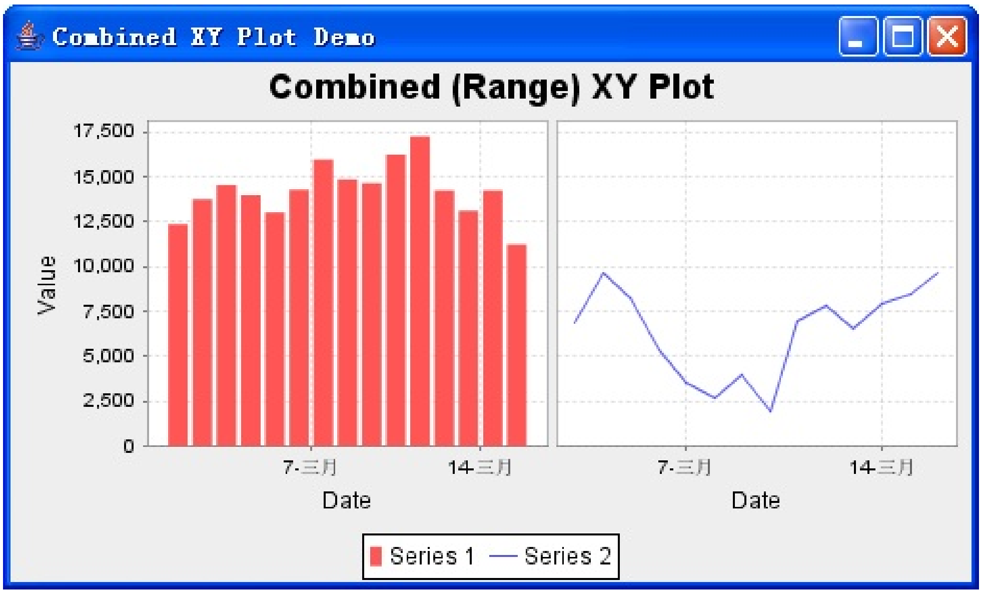


图14.4 组合Y-XY图区（参见：CombinedXYPlotDemo5.java）

图区可能水平显示也可能垂直显示（本例子中垂直显示）。

14.5.2 构建图表

CombinedXYPlotDemo2java实例演示了如何创建该类型的图表。关键的步骤是创建一个实例CombinedRangeXYPlot，并在该实例上添加两个子图区：

```
// create the plot...
CombinedRangeXYPlot plot = new CombinedRangeXYPlot(new NumberAxi
s("Value"));
plot.add(xyplot, 1);
plot.add(xyplot_0_, 1);
return new JFreeChart(
    "Combined (Range) XY Plot",
    JFreeChart.DEFAULT_TITLE_FONT, plot, true
);
```

注意两个图区的码值为什么都是1呢？因为该数值控制着每个图区分配的空间大小。

子图区是XYPlot实例，将自己的X轴设置为null。例如，下面的代码演示了这个特征：

```
IntervalXYDataset intervalxydataset = createDataset1();
XYBarRenderer xybarrenderer = new XYBarRenderer(0.2);
xybarrenderer.setBaseToolTipGenerator(new StandardXYToolTipGener
ator(
    "{0}: ({1}, {2})", new SimpleDateFormat("d-MMM-yyyy"),
    new DecimalFormat("0,000.0")));
XYPlot xyplot = new XYPlot(intervalxydataset, new DateAxis("Date
"),
    null, xybarrenderer);
XYDataset xydataset = createDataset2();
StandardXYItemRenderer standardxyitemrenderer = new StandardXYIt
emRenderer();
standardxyitemrenderer
    .setBaseToolTipGenerator(new StandardXYToolTipGenerator(
    "{0}: ({1}, {2})", new SimpleDateFormat("d-MMM-yyyy"),
    new DecimalFormat("0,000.0")));
XYPlot xyplot_0_ = new XYPlot(xydataset, new DateAxis("Date"), n
ull,
    standardxyitemrenderer);
```

15 数据源和JDBC(Dataset And JDBC)

15.1 简介

本章节，主要讲述使用JDBC从数据库表中获得数据的几种数据源方法：

- JDBCPieDataset
- JDBCCategoryDataset
- JDBCXYDataset

15.2 关于JDBC

JDBC（Java Data Base Connectivity,java数据库连接）是一种用于执行SQL语句的Java API，可以为多种关系数据库提供统一访问，它由一组用Java语言编写的类和接口组成。JDBC提供了一种基准，据此可以构建更高级的工具和接口，使数据库开发人员能够编写数据库应用程序。

15.3 样本数据

我们再看一下实际运行中的JDBC数据源。我们需要在一个测试数据库中创建一些样本数据。

下面列出了创建饼图、直方条形图和时序图的样本数据。

创建饼图可以使用下面数据（在表中称谓饼数据）：

CATEGORY	VALUE
London	54.3
New York	43.4
Paris	17.9

同样，直方条形图使用下面数据创建（在表中称谓种类数据）：

CATEGORY	SERIES1	SERIES2	SERIES3
London	54.3	32.1	53.4
New York	43.4	54.3	75.2
Paris	17.9	34.8	37.1

最后，时序图表的使用的数据如下（在表中称谓xy数据）：

X	SERIES1	SERIES2	SERIES3
1-Aug-2002	54.3	32.1	53.4
2-Aug-2002	43.4	54.3	75.2
3-Aug-2002	39.6	55.9	37.1
4-Aug-2002	35.4	55.2	27.5
5-Aug-2002	33.9	49.8	22.3
6-Aug-2002	35.2	48.4	17.7
7-Aug-2002	38.9	49.7	15.3
8-Aug-2002	36.3	44.4	12.1
9-Aug-2002	31.0	46.3	11.0

我们可以创建一个测试数据库，包含上面的表。这里我们创建一个jfreechartdb数据库。

在下一章节里，我们使用PostgreSQL创建数据。如果使用的是不同的数据库系统，我们需要对这个过程做一些修改。

15.4 PostgreSQL

15.4.1 关于PostgreSQL

PostgreSQL是一个非常强大的面向关系的数据库服务系统，是一个开源的分布式系统。我们可以从下面链接获得更多的信息：

http://www.postgresql.org

注意尽管PostgreSQL是开源的，但它具有其他大型商业关系数据库系统的大部分特征。这里鼓励你安装，并使用它。

15.4.2 创建一个新的数据库

首先，登录数据库管理系统，创建一个名为jfreechartdb的数据库。

```
CREATE DATABASE jfreechartdb;
```

其次，创建一个jfreechart用户：

```
CREATE USER jfreechart WITH PASSWORD 'password';
```

JDBC可以使用这个用户名和密码进行数据库的连接。

15.4.3 创建饼图数据

创建饼图数据库表：

```
CREATE TABLE piedata1 (
    category VARCHAR(32),
    value FLOAT
);
```

加入样本数据：

```
INSERT INTO piedata1 VALUES ('London', 54.3);
INSERT INTO piedata1 VALUES ('New York', 43.4);
INSERT INTO piedata1 VALUES ('Paris', 17.9);
```

15.4.4 创建种类图表数据

创建种类图数据库表：

```
CREATE TABLE categorydata1 (
    category VARCHAR(32),
    series1 FLOAT,
    series2 FLOAT,
    series3 FLOAT
);
```

加入样本数据：

```
INSERT INTO categorydata1 VALUES ('London', 54.3, 32.1, 53.4);
INSERT INTO categorydata1 VALUES ('New York', 43.4, 54.3, 75.2);
INSERT INTO categorydata1 VALUES ('Paris', 17.9, 34.8, 37.1);
```

15.4.5 创建XY图表数据

创建种类图数据库表：

```
CREATE TABLE xydata1 (
    date DATE,
    series1 FLOAT,
    series2 FLOAT,
    series3 FLOAT
);
```

加入样本数据：

```
INSERT INTO xydata1 VALUES ('1-Aug-2002', 54.3, 32.1, 53.4);
INSERT INTO xydata1 VALUES ('2-Aug-2002', 43.4, 54.3, 75.2);
INSERT INTO xydata1 VALUES ('3-Aug-2002', 39.6, 55.9, 37.1);
INSERT INTO xydata1 VALUES ('4-Aug-2002', 35.4, 55.2, 27.5);
INSERT INTO xydata1 VALUES ('5-Aug-2002', 33.9, 49.8, 22.3);
INSERT INTO xydata1 VALUES ('6-Aug-2002', 35.2, 48.4, 17.7);
INSERT INTO xydata1 VALUES ('7-Aug-2002', 38.9, 49.7, 15.3);
INSERT INTO xydata1 VALUES ('8-Aug-2002', 36.3, 44.4, 12.1);
INSERT INTO xydata1 VALUES ('9-Aug-2002', 31.0, 46.3, 11.0);
```

15.4.6 设置权限

最后一步是给样本数据授一读的权限给新用户jfreechart：

```
GRANT SELECT ON piedata1 TO jfreechart;
GRANT SELECT ON categorydata1 TO jfreechart;
GRANT SELECT ON xydata1 TO jfreechart;
```

15.5 JDBC驱动

为了使用JDBC访问样本数据，我们需要获得数据库的JDBC驱动。对于PostgreSQL，可以从下面的连接下载：

http://jdbc.postgresql.org

为了使用这个驱动，确保这个驱动jar文件加到classpath中。

15.6 应用演示

15.6.1 JDBC饼图演示

JDBC饼图演示实例将使用饼图数据表的数据产生饼图。该数据是由我们配置的数据库中获得的。

读数据的代码在方法readData()中：

```
private PieDataset readData() {
    JDBCPieDataset data = null;
    String url = "jdbc:postgresql://nomad/jfreechartdb";
    Connection con;
    try {
        Class.forName("org.postgresql.Driver");
    } catch (ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
    }
    try {
        con = DriverManager.getConnection(url, "jfreechart", "pa
ssword");
        data = new JDBCPieDataset(con);
        String sql = "SELECT * FROM PIEDATA1;";
        data.executeQuery(sql);
        con.close();
    } catch (SQLException e) {
        System.err.print("SQLException: ");
        System.err.println(e.getMessage());
    } catch (Exception e) {
        System.err.print("Exception: ");
        System.err.println(e.getMessage());
    }
    return data;
}
```

在代码中需要注意的事项：

- url是连接数据库的链接字符串。
- 返回的查询数据使用了JDBCPieDataset类对象进行了封装。详细内容见文档。

15.6.2 JDBC种类图演示

JDBC种类图应用使用种类数据产生了一个条形直方图。代码类似于JDBC饼图代码。但我们需要使用JDBCCategoryDataset类类封装格式化数据。

15.6.3 JDBC XY图演示

JDBC XY图应用使用XY数据产生了一个时序图。代码类似于JDBC饼图代码。但我们需要使用JDBCXYDataset类类封装格式化数据。

16 导出图表为PDF格式

16.1 简介

在本章节中，我们讲述如何将一个JFreeChart生成图表转换成PDF格式文件。主要使用的工具是IText。随着讲述，在后面的章节中用一个简单的实例说明创建PDF文件的过程，该PDF文件包含了一个简单的图表。生成的文件，可以使用Acrobat阅读器来阅读，也可以使用支持PDF文件阅读的阅读器来阅读。

16.2 什么是Acrobat PDF

Acrobat PDF是一款非常流行的电子文档阅读器。可实现不同的硬件平台和软件应用程序之间的信息共享,不受软件版本的不同和安装的字体的影响。PDF可以使用Adobe 提供的一个免费工具Acrodbat Reader来创建。Acrodbat Reader在终端用户各种平台上是有效的，包括GNU/Linux,Windows,Unix,Machintosh等。

如果你的系统上没有安装Acrodbat Reader，可以到下面链接去下载：

http://www.adobe.com/products/acrobat/readstep.html

16.3 IText

iText 是一个能够快速产生PDF文件的java类库。iText的主页下载地址是：

http://www.lowagie.com/iText

截止写本文的时间，最新的版本是2.0.6

16.4 Graphics2D

JFreeChart使用iText工具是非常容易的事情，因为iText提供了Graphics2D的实现。在我们说明实例应用之前，我们先回顾一下Graphics2D的类。

Java.awt. Graphics2D类，标准java2D API的一部分。定义了在二维空间中大量画文本和图形的方法。Graphics2D部分子类处理全部的转化细节，从输出（文本和图形）到具体设置的映射转化。

JFreeChart画图表时，仅仅使用Graphics2D定义的方法。这就意味着JFreeChart可以将图表输出到Graphics2D子类支持的任何设备。

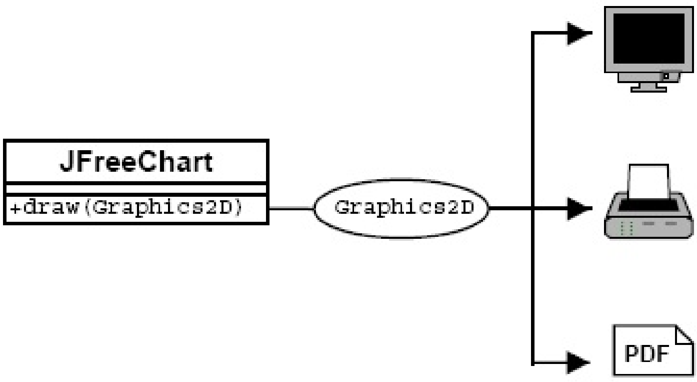


图16.2 JFreeChart画图的方法

iText工具融入了PdfGraphics2D的一个类，这就意味着iText使用Graphics2D类定义的方法产生PDF内容。并且正如你在后面的章节中看到的，在PDF格式中产生图表会变的非常的容易。

16.5 开始导出

为了完成和演示应用实例，我们需要下面的jar文件：

文件	描述
jfreechart-1.0.6.jar	JFreeChart类库
jcommon-1.0.9.jar	Jcommon类库
itext-2.0.6.jar	Itext类库

首先JFreeChart包括两个jar文件，其次iText需要一个jar文件。

16.6 实例应用

首先，需要创建一个图表，我们创建一个时序图，代码如下：

```
XYDataset dataset = createDataset();
JFreeChart chart = ChartFactory.createTimeSeriesChart(
    "Legal & General Unit Trust Prices",
    "Date",
    "Price Per Unit",
    dataset,
    true,
    true,
    false
);
```

这里没有任何的特殊代码——事实上，我们可以使用创建JFreeChart的其他对象替代上面的代码。

下一步，我们将在一个PDF文件中保存一个图表的副本：

```
File fileName = new File(System.getProperty("user.home") + "/jfr
eechart1.pdf");
saveChartAsPDF(fileName, chart, 400, 300, new DefaultFontMapper(
));
```

下面有一些需要注意的问题：

首先，PDF文件名称是硬编码完成的，不能修改。主要是在演示中，减少代码量。在实际应用中，我们需要提供一些让用户指定文件名称与路径的方式，比如弹出一个文件选择对话框。

其次，saveChartAsPDF()方法还未实现。为了创建这个方法，我们先创建另一个更通用的writeChartAsPDF().方法。该方法执行saveChartAsPDF()方法需要的全部工作。但该方法的输入参数是一个文件输出流而不是一个文件，代码如下：

```
public static void writeChartAsPDF(OutputStream out, JFreeChart chart,
int width, int height, FontMapper mapper) throws IOException {
    Rectangle pagesize = new Rectangle(width, height);
    Document document = new Document(pagesize, 50, 50, 50, 50);
    try {
        PdfWriter writer = PdfWriter.getInstance(document, out);
        document.addAuthor("JFreeChart");
        document.addSubject("Demonstration");
        document.open();
        PdfContentByte cb = writer.getDirectContent();
        PdfTemplate tp = cb.createTemplate(width, height);
        Graphics2D g2 = tp.createGraphics(width, height, mapper)
;
        Rectangle2D r2D = new Rectangle2D.Double(0, 0, width, he
ight);
        chart.draw(g2, r2D);
        g2.dispose();
        cb.addTemplate(tp, 0, 0);
    } catch (DocumentException de) {
        System.err.println(de.getMessage());
    }
    document.close();
}
```

在上面代码的方法里面，我们看到一些创建和代码iText文档的代码，从文档中获得了一个Graphics2D实例，使用Graphics2D对象画出这个图表，并关闭了这个文档。

同时我们也注意到方法的一个参数是FontMapper对象。iText使用FontMapper接口将java字体对象映射成基本的字体对象。DefaultFontMapper类预先默认映射为java本地化字体。如果你希望用这些字体，使用DefaultFontMapper构建缺省的对象即可，如果你相使用其他的字体（例如，支持一个特殊的字符集），那么我们需要做一些额外的工作。本章后面将有介绍。

在writeChartAsPDF()方法的实现里面，我们创建了一个自定义页面尺寸大小（匹配字符的需要尺寸）的PDF文档。我们提前设置了改变了字符的尺寸、位置并且在PDF文档中画出多个字符，以适应不同的页面尺寸。

现在我们将使用saveChartAsPDF()方法很容易的实现了将一个PDF数据发送到一个数据流上。建化了创建文件输出流的过程，并且将该对象传给了writeChartAsPDF()方法。代码如下：

```
public static void saveChartAsPDF(File file, JFreeChart chart, i
nt width,
    int height, FontMapper mapper) throws IOException {
    OutputStream out = new BufferedOutputStream(new FileOutputSt
ream(file));
    writeChartAsPDF(out, chart, width, height, mapper);
    out.close();
}
```

上面的每一步代码都是必须的。上面的代码组合成全部的代码如下（整个工程的代码都在这里，以便我们可以看到所有的声明和内容）：

```
package demo;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.text.SimpleDateFormat;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.time.Month;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
import org.jfree.data.xy.XYDataset;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Rectangle;
import com.lowagie.text.pdf.DefaultFontMapper;
import com.lowagie.text.pdf.FontMapper;
import com.lowagie.text.pdf.PdfContentByte;
import com.lowagie.text.pdf.PdfTemplate;
import com.lowagie.text.pdf.PdfWriter;
/**
* A simple demonstration showing how to write a chart to PDF for
mat using
* JFreeChart and iText.
* &lt;P&gt;
* You can download iText from http://www.lowagie.com/iText.
*/
public class PDFExportDemo1 {
    /**
    * Saves a chart to a PDF file.
    *
    * @param file
```

```
     * the file.
     * @param chart
     * the chart.
     * @param width
     * the chart width.
     * @param height
     * the chart height.
     */
    public static void saveChartAsPDF(File file, JFreeChart char
t, int width,
        int height, FontMapper mapper) throws IOException {
        OutputStream out = new BufferedOutputStream(new FileOutp
utStream(file));
        writeChartAsPDF(out, chart, width, height, mapper);
        out.close();
    }
    /**
     * Writes a chart to an output stream in PDF format.
     *
     * @param out
     * the output stream.
     * @param chart
     * the chart.
     * @param width
     * the chart width.
     * @param height
     * the chart height.
     *
     */
    public static void writeChartAsPDF(OutputStream out, JFreeCh
art chart,
    int width, int height, FontMapper mapper) throws IOException
 {
        Rectangle pagesize = new Rectangle(width, height);
        Document document = new Document(pagesize, 50, 50, 50, 5
0);
        try {
            PdfWriter writer = PdfWriter.getInstance(document, o
ut);
            document.addAuthor("JFreeChart");
            document.addSubject("Demonstration");
            document.open();
            PdfContentByte cb = writer.getDirectContent();
            PdfTemplate tp = cb.createTemplate(width, height);
            Graphics2D g2 = tp.createGraphics(width, height, map
per);
            Rectangle2D r2D = new Rectangle2D.Double(0, 0, width
, height);
            chart.draw(g2, r2D);
            g2.dispose();
            cb.addTemplate(tp, 0, 0);
        } catch (DocumentException de) {
            System.err.println(de.getMessage());
```

```
        }
        document.close();
    }
    /**
    * Creates a dataset, consisting of two series of monthly dat
a. * *
    *
    * @return the dataset.
    */
    public static XYDataset createDataset() {
        TimeSeries s1 = new TimeSeries("L&G European Index Trust
", Month.class);
        s1.add(new Month(2, 2001), 181.8);
        s1.add(new Month(3, 2001), 167.3);
        s1.add(new Month(4, 2001), 153.8);
        s1.add(new Month(5, 2001), 167.6);
        s1.add(new Month(6, 2001), 158.8);
        s1.add(new Month(7, 2001), 148.3);
        s1.add(new Month(8, 2001), 153.9);
        s1.add(new Month(9, 2001), 142.7);
        s1.add(new Month(10, 2001), 123.2);
        s1.add(new Month(11, 2001), 131.8);
        s1.add(new Month(12, 2001), 139.6);
        s1.add(new Month(1, 2002), 142.9);
        s1.add(new Month(2, 2002), 138.7);
        s1.add(new Month(3, 2002), 137.3);
        s1.add(new Month(4, 2002), 143.9);
        s1.add(new Month(5, 2002), 139.8);
        s1.add(new Month(6, 2002), 137.0);
        s1.add(new Month(7, 2002), 132.8);
        TimeSeries s2 = new TimeSeries("L&G UK Index Trust", Mon
th.class);
        s2.add(new Month(2, 2001), 129.6);
        s2.add(new Month(3, 2001), 123.2);
        s2.add(new Month(4, 2001), 117.2);
        s2.add(new Month(5, 2001), 124.1);
        s2.add(new Month(6, 2001), 122.6);
        s2.add(new Month(7, 2001), 119.2);
        s2.add(new Month(8, 2001), 116.5);
        s2.add(new Month(9, 2001), 112.7);
        s2.add(new Month(10, 2001), 101.5);
        s2.add(new Month(11, 2001), 106.1);
        s2.add(new Month(12, 2001), 110.3);
        s2.add(new Month(1, 2002), 111.7);
        s2.add(new Month(2, 2002), 111.0);
        s2.add(new Month(3, 2002), 109.6);
        s2.add(new Month(4, 2002), 113.2);
        s2.add(new Month(5, 2002), 111.6);
        s2.add(new Month(6, 2002), 108.8);
        s2.add(new Month(7, 2002), 101.6);
        TimeSeriesCollection dataset = new TimeSeriesCollection(
);
        dataset.addSeries(s1);
```

```
        dataset.addSeries(s2);
        return dataset;
    }
    public static void main(String[] args) {
        try {
            // create a chart...
            XYDataset dataset = createDataset();
            JFreeChart chart = ChartFactory.createTimeSeriesChar
t(
                "Legal & General Unit Trust Prices", "Date",
                "Price Per Unit", dataset, true, true, false);
            // some additional chart customisation here...
            XYPlot plot = chart.getXYPlot();
            XYLineAndShapeRenderer renderer = (XYLineAndShapeRen
derer) plot
                .getRenderer();
            renderer.setShapesVisible(true);
            DateAxis axis = (DateAxis) plot.getDomainAxis();
            axis.setDateFormatOverride(new SimpleDateFormat("MMM
-yyyy"));
            // write the chart to a PDF file...
            File fileName = new File(System.getProperty("user.ho
me")
                + "/jfreechart1.pdf");
            System.out.println(fileName.getPath());
            saveChartAsPDF(fileName, chart, 400, 300, new Defaul
tFontMapper());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

在你完成和运行上面的应用之前，记得修改PDF文件的名称以满足我们的要求。同时前面16.5节提到的jar也必须在我们的classpath中。.

16.7 查看PDF 文件

在我们完成上面实例，运行实例，会产生一个PDF文档。我们可以使用一个PDF浏览器（比如AcrobatReader（或者其他支持的阅读器，Gnome PDF Viewer））查看该文件，显示的界面如下图16.3

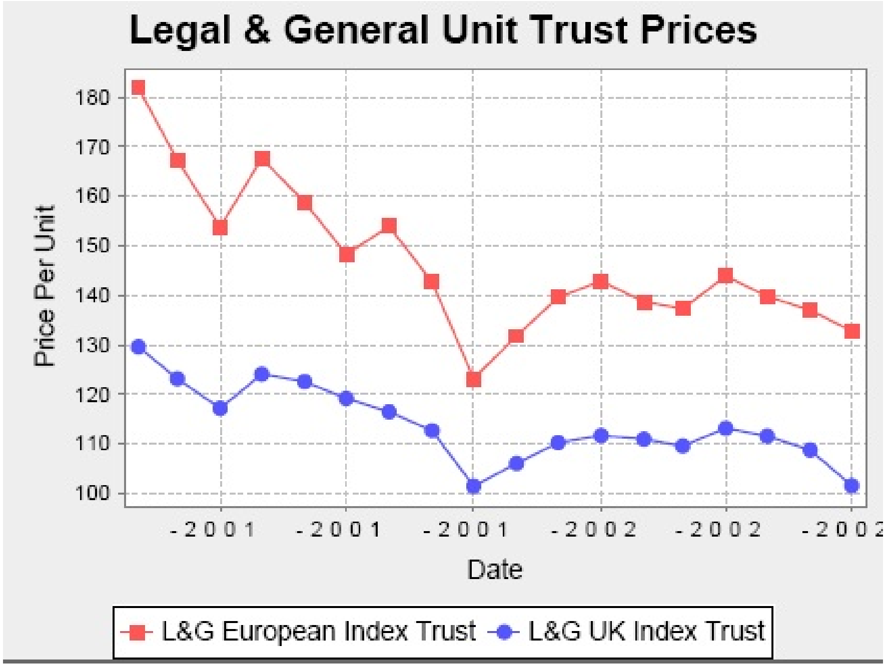


图16.3 JFreeChart使用iText生成的PDF文件图

大部分的PDF阅读器都提供了缩放技术，以允许我们更进一步浏览我们的图表。

16.8 Unicode字符问题

声明：由于本人对自字符集了解不深，因此翻译效果比较差，忘各大网友给予大力支持。

在我们关心我们所使用的字体字符集时，在JFreeChart和iText中使用Unicode字符集是没有任何问题的。在上面的例子中我们需要做一些修改来演示如何做到这些。

16.8.1 背景

Java使用同一的字符集译码成文本字符串。这种译码对每个字符使用16进制。这就意味着将有65，536个有效的不同字符集（在Unicode标准中定义了大约38，000个字符）。

我们可以在JFreeChart和iText中使用这些字符,但归于一条：那就是只要我们使用的字体，包括我们用来显示的文本或者不显示的，都必须定义这些字符。许多字体并不完全显示成Unicode字符集。下面的网站含有那些的确支持Unicode的字体的有用信息：

http://www.slovo.info/unifonts.htm

我们可以成功的提取使用tahoma.ttf字体。实际上，下面实例中我们将使用该字体，Tahoma字体并不是支持Unicode定义的每一个字符。因此，如果我们想使用一种特殊的字体时，就得必须选Unicode中一种相近的字体来代替。我们系统上都安装了字体Unicode MS (arialuni.ttf)——该字体完全支持Unicode字符集，尽管这种字体的定义的文件特别大（大约24M）

16.8.2 字体、iText和Java

iText依照PDF规格来处理字体，这就对使用PDF文件嵌入的字体来处理文件带来了非常大的方便性，同时也需要自由读取定义文件的字体。

而java在字体类中汲取了部分字体格式的大部分细节内容。

在iText中，为支持Graphics2D的实现画图功能，实现从Java字体对象到BaseFont对象的映射字体对象是非常有必要的。这就是FontMapper接口所扮演的角色。

如果我们使用缺省的构建器构建了一个新的DefaultFontMapper实例，那么总会带有Java规格定义的本地字体映射。但是如果我们想使用其他一些字体——并且我们必须使用Unicode之外的字符——那么我们需要将其他的字符映射加入到DefaultFontMapper对象中。

16.8.3 映射第三方的字体

这里我们决定使用Tahoma字体来显示标题。该字体的定义文件(tahoma.ttf)在我们系统下面的目录下可以找到。

```
/opt/sun-jdk-1.4.2.08/jre/lib/fonts
```

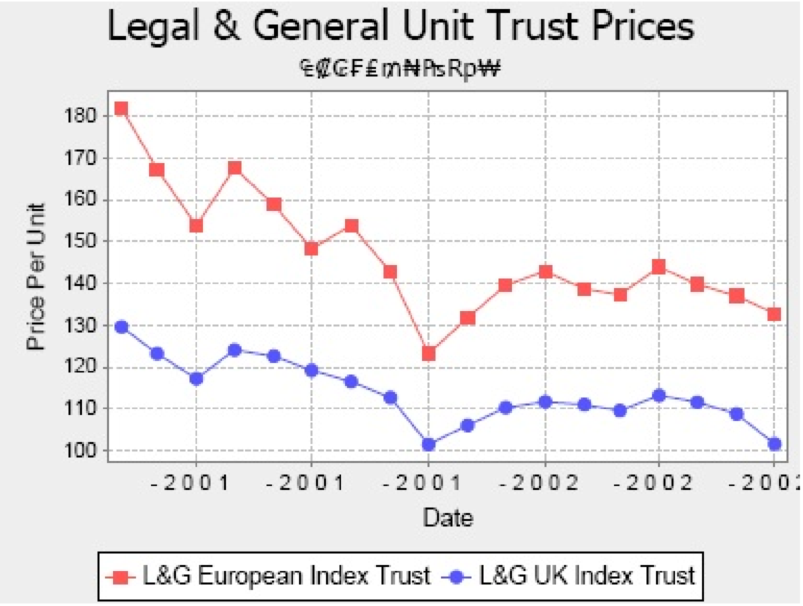
现在我们使用代码说明，使用iText创建FontMapper对象来使用Tahoma字体：

```
//设置字体
DefaultFontMapper mapper = new DefaultFontMapper();
mapper.insertDirectory("D:\\jre1.5.0_10\\lib\\fonts");
DefaultFontMapper.BaseFontParameters pp =
    mapper.getBaseFontParameters("Tahoma");
if (pp!=null) {
    pp.encoding = BaseFont.IDENTITY_H;
}
```

现在我们可以修改创建图表的代码，将图表的标题使用该字体：

```
TextTitle textTitle = chart.getTitle();
textTitle.setFont(new Font("Tahoma", Font.PLAIN, 20));
String text = "\u278A\u20A0\u20A1\u20A2\u20A3\u20A4\u20A5\u20A6\
u20A7\u20A8\u20A9";
// String text = "hi";
Font font = new Font("Tahoma", Font.PLAIN, 12);
TextTitle subtitle = new TextTitle(text, font);
chart.addSubtitle(subtitle);
```

副标题的输出如下图16.2所示。实例已经嵌入到PDF文件中。因此本文演示的该小程序很好的展示了这种类型的输出，给出了详细的步骤指南，便于我们正确使用。



如图16.2 一个Unicode副标题的图表

17 导出图表为SVG格式

17.1 简介

在本章里，我们介绍了一个简单实例，实例演示使用JFreeChart和Batik工具（SVG开源的类库）如何将图表导出为SVG格式。

17.2 背景

17.2.1 什么是SVG?

SVG(可放缩的矢量图形) 是W3C(World Wide Web ConSor—tium国际互联网标准组织)在2000年8月制定的一种基于XML格式的新的二维矢量图形格式，也是规范中的网络矢量图形标准。

17.2.2 Batik

Batik是一个java开源的工具包，允许我们产生SVG内容。可以从下面的链接获得有效的Batik：

http://xml.apache.org/batik

在写本文之前，Batik最新的版本是1.7

17.3 实例代码

17.3.1 JFreeChart 和Batik

JFreeChart和Batik兼容性非常好，因为：

- 因为JFreeChart画的所有图表的输出都是使用的Java的Graphics2D；
- Batik具体实现了Graphics2D产生SVG输出的功能（SVGGraphics2D）。

在本章节，使用一个简单的实例说明使用JFreeChart和Batik实现SVG的输出。关于该实例的详细技术详见下面链接：

http://xml.apache.org/batik/svggen.html

17.3.2 开始

首先，我们需要下载Batik并依照网站的指导进行安装。

确保下章节的例子能够正常运行，需要将下面的jar包加到我们的classpath：

文件	描述
jcommon-1.0.9.jar	JFreeChart的通用类包。
jfreechart-1.0.6.jar	JFreeChart的类包
batik-awt-util.jar	Batik实时运行文件
batik-dom.jar	Batik实时运行文件
batik-svggen.jar	Batik实时运行文件
batik-util.jar	Batik实时运行文件

17.3.3 实例应用

在我们的开发环境中创建一个工程，并且将上节列出的jar包添加到工程路径上，并输入下面代码：

```
package demo;
import java.awt.geom.Rectangle2D;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
import org.apache.batik.dom.GenericDOMImplementation;
import org.apache.batik.svggen.SVGGraphics2D;
```

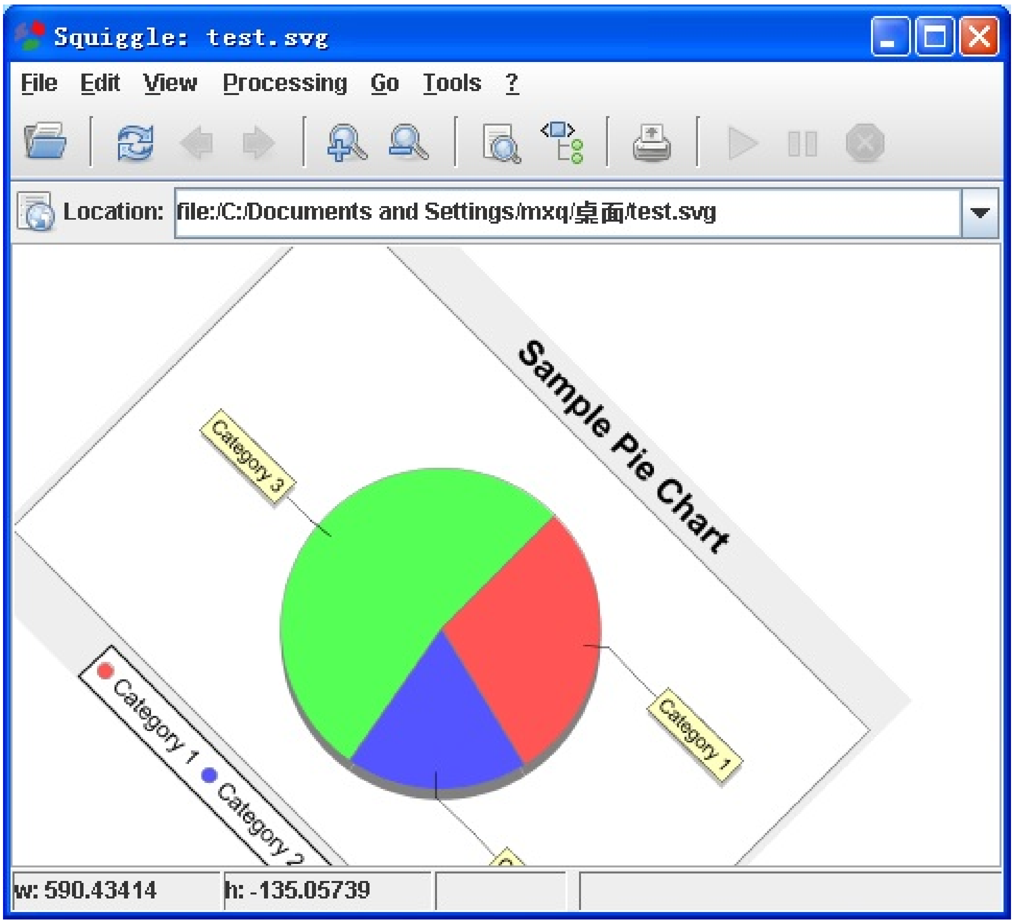
```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.data.general.DefaultPieDataset;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
/**
* A demonstration showing the export of a chart to SVG format.
*/
    public class SVGExportDemo {
    /**
    * Starting point for the demo.
    *
    * @param args
    * ignored.
    */
    public static void main(String[] args) throws IOException {
        // create a dataset...
        DefaultPieDataset data = new DefaultPieDataset();
        data.setValue("Category 1", new Double(43.2));
        data.setValue("Category 2", new Double(27.9));
        data.setValue("Category 3", new Double(79.5));
        // create a chart
        JFreeChart chart = ChartFactory.createPieChart("Sample P
ie Chart",
            data, true, false, false);
        // THE FOLLOWING CODE BASED ON THE EXAMPLE IN THE BATIK
DOCUMENTATION...
        // Get a DOMImplementation
        DOMImplementation domImpl = GenericDOMImplementation
            .getDOMImplementation();
        // Create an instance of org.w3c.dom.Document
        Document document = domImpl.createDocument(null, "svg",
null);
        // Create an instance of the SVG Generator
        SVGGraphics2D svgGenerator = new SVGGraphics2D(document)
;
        // set the precision to avoid a null pointer exception i
n Batik 1.5
        svgGenerator.getGeneratorContext().setPrecision(6);
        // Ask the chart to render into the SVG Graphics2D imple
mentation
        chart.draw(svgGenerator, new Rectangle2D.Double(0, 0, 40
0, 300), null);
        // Finally, stream out SVG to a file using UTF-8 charact
er to
        // byte encoding
        boolean useCSS = true;
        Writer out = new OutputStreamWriter(new FileOutputStream
(new File(
            "test.svg")), "UTF-8");
        svgGenerator.stream(out, useCSS);
    }
}
```

17.3.4 浏览SVG图

Batik类库内包含了一个"Squiggle"的小应用，我们可以使用该工具浏览SVG文件。我们可以使用下面命令打开：

```
java -jar batik-squiggle.jar
```

下图截屏显示了上述代码创建的一个饼图的界面。使用应用浏览器工具，将图表在浏览器中进行了45度旋转。



如图17.1 SVG截图

18 Applet

18.1 简介

局限于一些条件，在Applet中使用JFreeChart还是比较容易的。本章节对Applet进行了整体的介绍，并举例说明了工作过程。这样对我们开始使用Applet提供极大帮助。

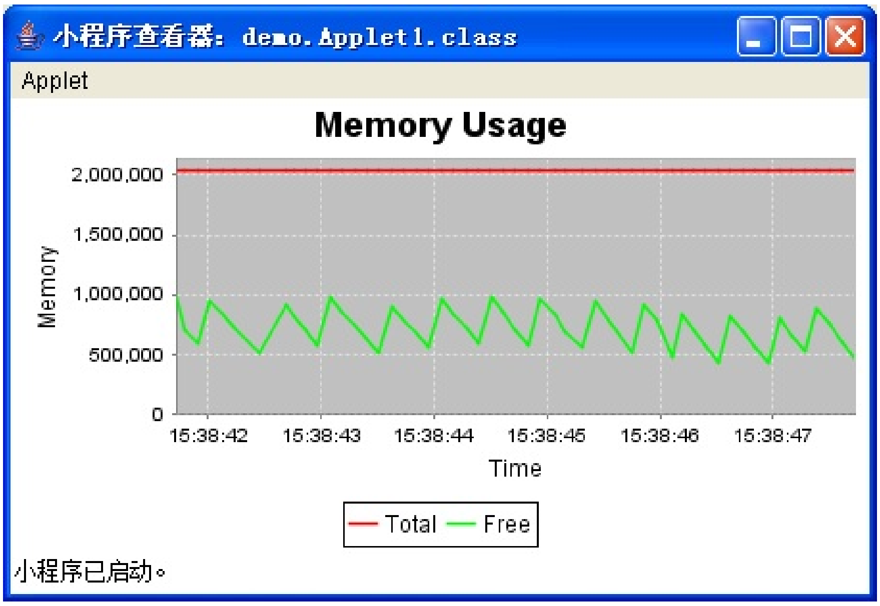


图18.1 JFreeChart在Applet上的应用

图18.1显示了一个使用JFreeChart的Applet简单应用。该applet可以通过下面链接看效果。

http://www.object-refinery.com/jfreechart/applet.html

后面的章节有全部的代码。

18.2 问题

在开发applet时，考虑的主要问题（与JFreeChart无关）是：

- 浏览器器支持
- 安全约束
- 字节码文件大小

在我们用提供的有效资源写applets时，确保我们对上面问题有所了解

18.2.1 浏览器支持

绝大部分的web浏览器均对最新版本的JDK1.5提供支持，因此使用JFreeChart运行applets也是绝对没有任何问题的（JFreeChart可以运行在JDK1.3.1版本或以上版本）。尽管如此，很大一部分用户还是使用一个浏览器的——微软的IE浏览器——该浏览器仅仅支持JDK1.1版本，并且现在已经过期。这里有一个问题是，使用JFreeChart的applet应用在微软的IE上是不能默认运行的。这就必须下载一个Java的插件，但是这样会造成很多不必要的麻烦和困难，最终的问题就是那些开发者选择写applets开发的问题，这导致开发者放弃开发applets，而选择Java Servlets（见下一章节）。

18.2.2 安全

Applets设计时，是符合java安全规范的。当一个applet运行在我们的web浏览器上时，在操作上是受到很大的约束的。例如，一个applet典型的是不能读写本地文件系统的。关于Java安全机制的描述已经超出了本章的范围，但是我们必须意识到JFreeChart的一些功能在applets上是不能运行的（例如将图表保存成PNG格式的文件），主要受java缺省的安全规则约束。如果我们想使用这些功能，那么我们需要认真学习一下java的安全机制的更多细节。

18.2.3 代码大小

最后一个文件就是我们applet运行时需要的代码量问题。在我们运行一个applet之前，代码被下载到本地客户端。显然对用户来说是有带宽限制的，代码量的大小成了关键问题。JFreeChart代码的jar文件大约是1M左右，对JFreeChart支持的图表来说，不算很大，但对使用modem拨号上网的用户来说，的确不是很理想的。同时我们需要将JCommon的jar包（大约290KB）加到我们的applet上。考虑到这些问题，我们将对JFreeChart进行重新打包，仅仅将applet需要的类文件包含进来，从而优化代码结构。

18.3 实例应用

正如在简介中所提及的，使用JFreeChart的applet可以在下面链接中看到：

http://www.object-refinery.com/jfreechart/applet.html

运行applet应用时，需要两个方面饿支持。一是代码方式创建applet，二是HTML文件用来调用applet。

18.3.1 HTML

因为applet需要引入额外的jar文件，所以在HTML中使用applet是显得非常重要。HTML applet标签如下：

```
<APPLET ARCHIVE="jfreechart-1.0.6-applet-demo.jar,
jfreechart-1.0.6.jar,jcommon-1.0.9.jar"
CODE="demo.applet.Applet1" width=640 height=260
ALT="You should see an applet, not this text.">
</APPLET>
```

注意这里有三个jar需要引入，第一个包含了applet类，另外两个jar文件是JFreeChart和JCommon类库。我们需要在HTML文件中将applet标签引入。

18.3.2 源代码

实例applet的源代码见下（代码中我们使用了很少的applet特殊的代码，仅仅扩展了JApplet）：

```
package demo;
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JApplet;
import javax.swing.Timer;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.axis.DateAxis;
import org.jfree.chart.axis.NumberAxis;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYItemRenderer;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.time.Millisecond;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
/**
```

```
 * A simple applet demo.
 */
public class Applet1 extends JApplet {
    /** Time series for total memory used. */
    private TimeSeries total;
    /** Time series for free memory. */
    private TimeSeries free;
    /**
    * Creates a new instance.
    */
    public Applet1() {
        // create two series that automatically discard data mor
e than
        // 30 seconds old...
        this.total = new TimeSeries("Total", Millisecond.class);
        this.total.setMaximumItemAge(30000);
        this.free = new TimeSeries("Free", Millisecond.class);
        this.free.setMaximumItemAge(30000);
        TimeSeriesCollection dataset = new TimeSeriesCollection(
);
        dataset.addSeries(total);
        dataset.addSeries(free);
        DateAxis domain = new DateAxis("Time");
        NumberAxis range = new NumberAxis("Memory");
        XYItemRenderer renderer = new XYLineAndShapeRenderer(tru
e, false);
        XYPlot plot = new XYPlot(dataset, domain, range, rendere
r);
        plot.setBackgroundPaint(Color.lightGray);
        plot.setDomainGridlinePaint(Color.white);
        plot.setRangeGridlinePaint(Color.white);
        renderer.setSeriesPaint(0, Color.red);
        renderer.setSeriesPaint(1, Color.green);
        renderer.setSeriesStroke(0, new BasicStroke(1.5f));
        renderer.setSeriesStroke(1, new BasicStroke(1.5f));
        domain.setAutoRange(true);
        domain.setLowerMargin(0.0);
        domain.setUpperMargin(0.0);
        domain.setTickLabelsVisible(true);
        range.setStandardTickUnits(NumberAxis.createIntegerTickU
nits());
        JFreeChart chart = new JFreeChart("Memory Usage",
        JFreeChart.DEFAULT_TITLE_FONT, plot, true);
        chart.setBackgroundPaint(Color.white);
        ChartPanel chartPanel = new ChartPanel(chart);
        chartPanel.setPopupMenu(null);
        getContentPane().add(chartPanel);
        new Applet1.DataGenerator().start();
    }
    /**
    * Adds an observation to the ’total memory’ time series.
    *
    * @param y
```

```
     * the total memory used.
     */
     private void addTotalObservation(double y) {
         total.add(new Millisecond(), y);
     }
     /**
     * Adds an observation to the ’free memory’ time series.
     *
     * @param y
     * the free memory.
     */
     private void addFreeObservation(double y) {
         free.add(new Millisecond(), y);
     }
     /**
     * The data generator.
     */
     class DataGenerator extends Timer implements ActionListener
 {
         /**
         * Constructor.
         */
         DataGenerator() {
             super(100, null);
             addActionListener(this);
         }
         /**
         * Adds a new free/total memory reading to the dataset.
         *
         * @param event
         * the action event.
         */
         public void actionPerformed(ActionEvent event) {
             long f = Runtime.getRuntime().freeMemory();
             long t = Runtime.getRuntime().totalMemory();
             addTotalObservation(t);
             addFreeObservation(f);
         }
     }
 }
```

19 Servlets

19.1 介绍

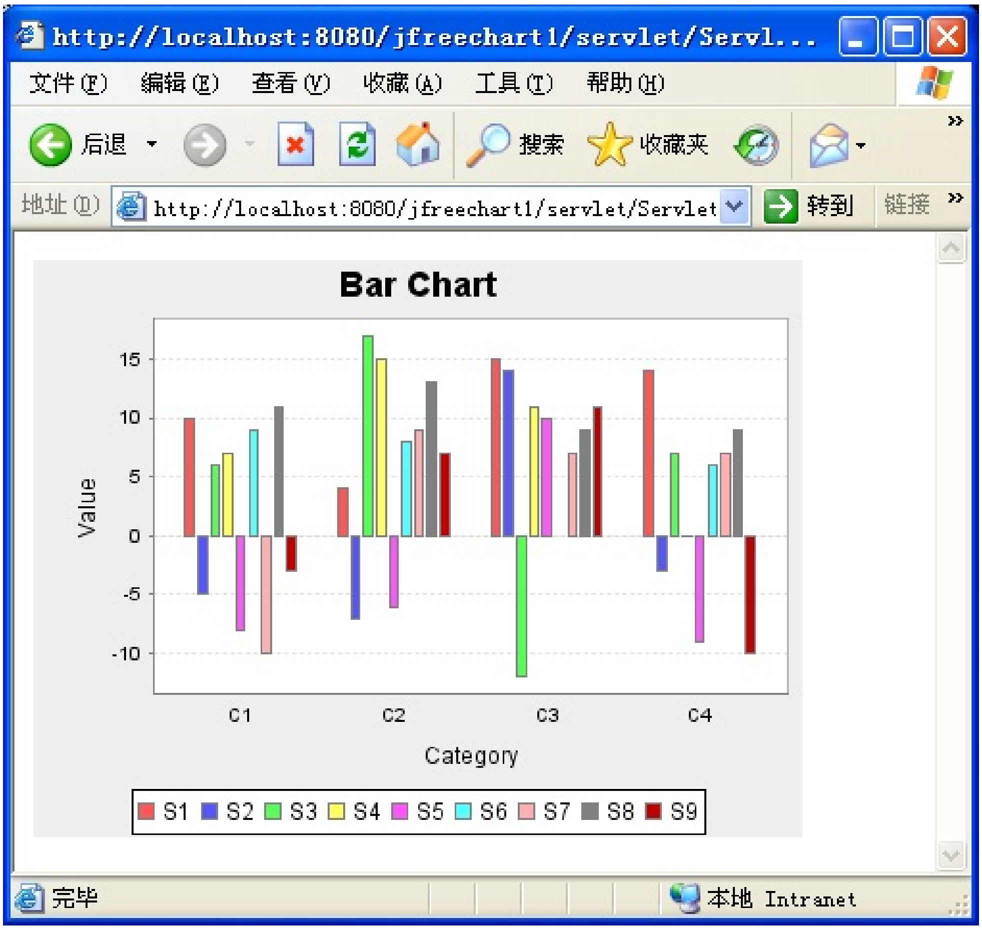
Java Servlet API是一套创建web应用非常流行成熟的技术。在servlet环境中使用JFreeChart是非常合适的。在本章节中，协助开发者在web应用中使用JFreeChart。本章所有的实例可以从下面链接中下载：

http://www.object-refinery.com/jfreechart/premium/index.html

下载的文件名为：jfreechart-1.0.6-demo.zip（该信息是收费的）.

19.2 编写一个简单的Servlet应用

ServletDemo1类实现了一个非常简单的servlet，该servlet返回了一个PNG图，PNG图是使用JFreeChart生成的直方条形图表。当运行该程序时，servlet在客户端仅仅显示一幅图片，而没有任何的HTML修饰，参见下图19.1。



如图19.1 浏览器中的servlet效果

我们以这种方式显示图片，是没有特殊意义，仅仅是为了：

- 很好演示servlets的请求—响应交互特征；
- 作为测试实例非常有用，我们会了解如何配置一个服务环境，如何让页面控件工作。

我们可以浏览后面更复杂的实例，显示使用HMTL表单如何请求不同的图表，并且将产生的图表的输出植入到HTML中。下面是基本servlet的代码。

```
package demo;
import java.io.IOException;
import java.io.OutputStream;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
/**
* A basic servlet that returns a PNG image file generated by JFr
eeChart. This
* class is described in the JFreeChart Developer Guide in the "S
ervlets"
* chapter.
*/
public class ServletDemo1 extends HttpServlet {
    /**
    * Creates a new demo.
    */
    public ServletDemo1() {
    // nothing required
    }
    /**
    * Processes a GET request.
    *
    * @param request
    * the request.
    * @param response
    * the response.
    *
    * @throws ServletException
    * if there is a servlet related problem.
    * @throws IOException
    * if there is an I/O problem.
    */
    public void doGet(HttpServletRequest request, HttpServletRes
ponse response)
    throws ServletException, IOException {
        OutputStream out = response.getOutputStream();
        try {
            DefaultCategoryDataset dataset = new DefaultCategory
Dataset();
            dataset.addValue(10.0, "S1", "C1");
            dataset.addValue(4.0, "S1", "C2");
            dataset.addValue(15.0, "S1", "C3");
            dataset.addValue(14.0, "S1", "C4");
            dataset.addValue(-5.0, "S2", "C1");
            dataset.addValue(-7.0, "S2", "C2");
            dataset.addValue(14.0, "S2", "C3");
            dataset.addValue(-3.0, "S2", "C4");
            dataset.addValue(6.0, "S3", "C1");
            dataset.addValue(17.0, "S3", "C2");
            dataset.addValue(-12.0, "S3", "C3");
            dataset.addValue(7.0, "S3", "C4");
            dataset.addValue(7.0, "S4", "C1");
            dataset.addValue(15.0, "S4", "C2");
```

```
            dataset.addValue(11.0, "S4", "C3");
            dataset.addValue(0.0, "S4", "C4");
            dataset.addValue(-8.0, "S5", "C1");
            dataset.addValue(-6.0, "S5", "C2");
            dataset.addValue(10.0, "S5", "C3");
            dataset.addValue(-9.0, "S5", "C4");
            dataset.addValue(9.0, "S6", "C1");
            dataset.addValue(8.0, "S6", "C2");
            dataset.addValue(null, "S6", "C3");
            dataset.addValue(6.0, "S6", "C4");
            dataset.addValue(-10.0, "S7", "C1");
            dataset.addValue(9.0, "S7", "C2");
            dataset.addValue(7.0, "S7", "C3");
            dataset.addValue(7.0, "S7", "C4");
            dataset.addValue(11.0, "S8", "C1");
            dataset.addValue(13.0, "S8", "C2");
            dataset.addValue(9.0, "S8", "C3");
            dataset.addValue(9.0, "S8", "C4");
            dataset.addValue(-3.0, "S9", "C1");
            dataset.addValue(7.0, "S9", "C2");
            dataset.addValue(11.0, "S9", "C3");
            dataset.addValue(-10.0, "S9", "C4");
            JFreeChart chart = ChartFactory.createBarChart("Bar
Chart",
                "Category", "Value", dataset, PlotOrientation.VE
RTICAL,
                true, true, false);
            response.setContentType("image/png");
            ChartUtilities.writeChartAsPNG(out, chart, 400, 300)
;
        } catch (Exception e) {
            System.err.println(e.toString());
        } finally {
            out.close();
        }
    }
}
```

当一个客户端（通常是一个web浏览器）发出一个请求时，Servlet引擎调用DoGet()方法，以响应这个请求，servlet执行下面几步：

- 为客户端返回的输出获得一个输出流引用
- 创建一个图表；
- 响应的内容类型设置为image/png，这告诉客户端接受的数据类型是什么；
- 一个图表的PNG图表beii写进输出流；
- 输出流关闭。

19.3 编译实例Servlet

注意在javax.servlet.*包（包括子包）内的类，也就是实例使用的servlet，并不是J2SE的一部分。为了使用J2SE编译上面的代码需要另一个jar文件servlet.jar。我们使用了tomcat（是用java编写的一个开源servlet引擎）版本下面的这个servlet.jar文件。Tomcat可以在下面链接中获得：

http://tomcat.apache.org/

同时我们需要JFreeChat和JCommon两个类包文件来编译上面的代码。改变我们当前的工作目录，输入下面的命令（如果在Windows上，你需要将冒号“：“更改为分号“；“）

```
javac -classpath jfreechart-1.0.6.jar:lib/jcommon-1.0.9.jar:lib/
servlet.jar
source/demo/ServletDemo1.java
```

这样就生成了一个ServletDemo1.class文件，下一章内容描述了如何使用Tomcat部署这个servlet。

19.4 部署实例Servlet

Servlets部署在我们服务引擎的webapps的目录下面，在我们的例子中，使用的是Tomcat 5.5.20，将代码部署在：

```
D:\apache-tomcat-5.5.20\webapps\jfreechart1
```

在webapp目录下，创建一个目录jfreechart1来存放servlet演示实例，然后创建下面的结构目录。

```
.../jfreechart1/WEB-INF/web.xml
.../jfreechart1/WEB-INF/lib/jfreechart-1.0.6.jar
.../jfreechart1/WEB-INF/lib/jcommon-1.0.9.jar
.../jfreechart1/WEB-INF/classes/demo/ServletDemo1.class
```

我们需要创建web.xml文件——提供servlet的信息。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
    <servlet>
        <servlet-name>ServletDemo1</servlet-name>
        <servlet-class>demo.ServletDemo1</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletDemo1</servlet-name>
        <url-pattern>/servlet/ServletDemo1</url-pattern>
    </servlet-mapping>
</web-app>
```

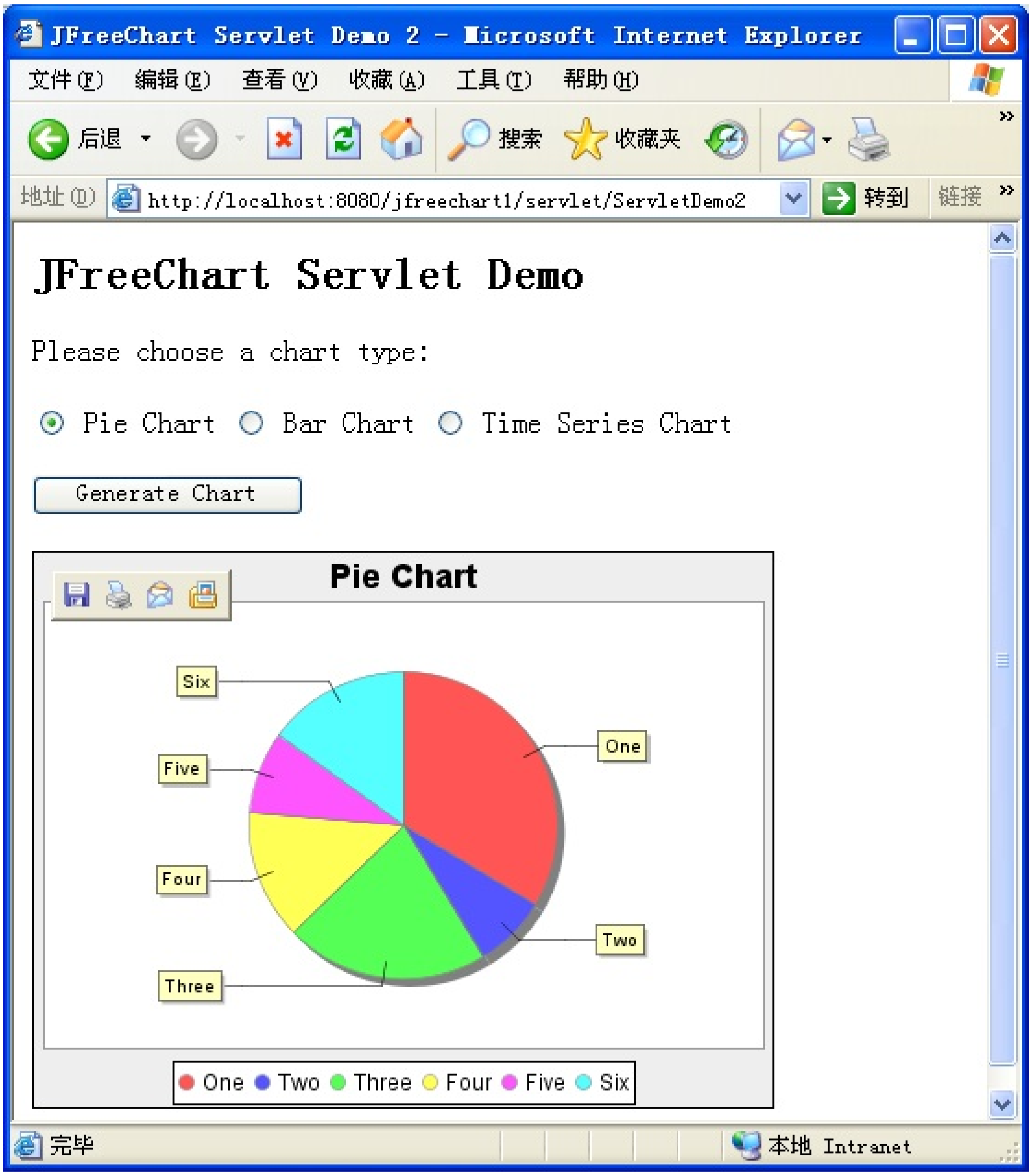
一旦上面的文件部署在servlet服务引擎上，然后启动我们的servlet服务引擎，在web浏览器中输入下面的地址：

http://localhost:8080/jfreechart1/servlet/ServletDemo1

如果不出现意外，我们就会在浏览器中看到如图19.1所示的界面。

19.5 在HMTL页面种嵌入图表

在HTML页面中嵌入servlet产生的图表图像也是可以的，下面实例ServletDemo2演示了这一特征。ServletDemo2实例处理一个HTML页面的请求，该HTML引用了另一个servlet（ServletDemo2ChartGenerator），引用的这个servlet返回了一个图表产生的PNG图像。最终的结果就是将图表嵌入到了一个HTML中，如图19.2所示：



如图19.2 浏览器中的ServletDemo2

全部代码如下：

```
package demo;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
* A basic servlet that generates an HTML page that displays a ch
art generated
* by JFreeChart.
* &lt;P&gt;
* This servlet uses another servlet (ServletDemo2ChartGenerator)
 to create a
* PNG image for the embedded chart.
* &lt;P&gt;
* This class is described in the JFreeChart Developer Guide.
*/
public class ServletDemo2 extends HttpServlet {
    /**
    *
    */
    private static final long serialVersionUID = 902404046769790
9853L;
    /**
    * Creates a new servlet demo.
    */
    public ServletDemo2() {
        // nothing required
    }
    /**
    * Processes a POST request.
    * &lt;P&gt;
    * The chart.html page contains a form for generating the fir
st request,
    * after that the HTML returned by this servlet contains the
same form for
    * generating subsequent requests.
    *
    * @param request
    * the request.
    * @param response
    * the response.
    *
    * @throws ServletException
    * if there is a servlet related problem.
    * @throws IOException
    * if there is an I/O problem.
    */
    public void doPost(HttpServletRequest request, HttpServletRe
sponse response)
        throws ServletException, IOException {
        PrintWriter out = new PrintWriter(response.getWriter());
        try {
            String param = request.getParameter("chart");
            response.setContentType("text/html");
            out.println("<HTML>");
            out.println("<HEAD>");
```

```
            out.println("<TITLE>JFreeChart Servlet Demo 2</TITLE
>");
            out.println("</HEAD>");
            out.println("<BODY>");
            out.println("<H2>JFreeChart Servlet Demo</H2>");
            out.println("<P>");
            out.println("Please choose a chart type:");
            out.println("<FORM ACTION=\"ServletDemo2\" METHOD=PO
ST>");
            String pieChecked = (param.equals("pie") ? " CHECKED
" : "");
            String barChecked = (param.equals("bar") ? " CHECKED
" : "");
            String timeChecked = (param.equals("time") ? " CHECK
ED" : "");
            out.println("<INPUT TYPE=\"radio\" NAME=\"chart\" VA
LUE=\"pie\""
                + pieChecked + "> Pie Chart");
            out.println("<INPUT TYPE=\"radio\" NAME=\"chart\" VA
LUE=\"bar\""
                + barChecked + "> Bar Chart");
            out.println("<INPUT TYPE=\"radio\" NAME=\"chart\" VA
LUE=\"time\""
                + timeChecked + "> Time Series Chart");
            out.println("<P>");
            out.println("<INPUT TYPE=\"submit\" VALUE=\"Generate
 Chart\">");
            out.println("</FORM>");
            out.println("<P>");
            out.println("<IMG SRC=\"ServletDemo2ChartGenerator?t
ype=" + param
                + "\" BORDER=1 WIDTH=400 HEIGHT=300/>");
            out.println("</BODY>");
            out.println("</HTML>");
            out.flush();
            out.close();
        } catch (Exception e) {
            System.err.println(e.toString());
        } finally {
            out.close();
        }
    }
}
```

注意该代码是如何从响应的参数获得一个引用的，而不是上面实例中的一个输出流。愿意是因为该servlet将返回HTML文本，与前章返回的二进制数据（一个PNG图片）不同。响应的类型设置成为text/html格式，因为servlet返回的是HTML文件。重要的一点就是HTML引用另一个servlet（ServletDemo2ChartGenerator）中的<IMG>标签，ServletDemo2ChartGenerator创建了必要的图表图片。HTML使用<FORM>元素来建立图表参数控制着实际图表的返回。

下面是ServletDemo2ChartGenerator的全部代码：

```
package demo;
import java.io.IOException;
import java.io.OutputStream;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartUtilities;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.time.Day;
import org.jfree.data.time.TimeSeries;
import org.jfree.data.time.TimeSeriesCollection;
import org.jfree.data.xy.XYDataset;
import org.jfree.date.SerialDate;
/**
* A servlet that returns one of three charts as a PNG image file
. This servlet
* is referenced in the HTML generated by ServletDemo2.
* &lt;P&gt;
* Three different charts can be generated, controlled by the ’ty
pe’ parameter.
* The possible values are ’pie’, ’bar’ and ’time’ (for time seri
es).
* &lt;P&gt;
* This class is described in the JFreeChart Developer Guide.
*/
public class ServletDemo2ChartGenerator extends HttpServlet {
    /**
    * Default constructor.
    */
    public ServletDemo2ChartGenerator() {
        // nothing required
    }
    /**
    * Process a GET request.
    *
    * @param request
    * the request.
    * @param response
    * the response.
    *
    * @throws ServletException
    * if there is a servlet related problem.
    * @throws IOException
    * if there is an I/O problem.
    */
```

```
    public void doGet(HttpServletRequest request, HttpServletRes
ponse response)
    throws ServletException, IOException {
        OutputStream out = response.getOutputStream();
        try {
            String type = request.getParameter("type");
            JFreeChart chart = null;
            if (type.equals("pie")) {
                chart = createPieChart();
            } else if (type.equals("bar")) {
                chart = createBarChart();
            } else if (type.equals("time")) {
                chart = createTimeSeriesChart();
            }
            if (chart != null) {
                response.setContentType("image/png");
                ChartUtilities.writeChartAsPNG(out, chart, 400,
300);
            }
        } catch (Exception e) {
            System.err.println(e.toString());
        } finally {
            out.close();
        }
    }
    /**
    * Creates a sample pie chart.
    *
    * @return a pie chart.
    */
    private JFreeChart createPieChart() {
        // create a dataset...
        DefaultPieDataset data = new DefaultPieDataset();
        data.setValue("One", new Double(43.2));
        data.setValue("Two", new Double(10.0));
        data.setValue("Three", new Double(27.5));
        data.setValue("Four", new Double(17.5));
        data.setValue("Five", new Double(11.0));
        data.setValue("Six", new Double(19.4));
        JFreeChart chart = ChartFactory.createPieChart("Pie Char
t", data, true,
            true, false);
        return chart;
    }
    /**
    * Creates a sample bar chart.
    *
    * @return a bar chart.
    */
    private JFreeChart createBarChart() {
        DefaultCategoryDataset dataset = new DefaultCategoryData
set();
        dataset.addValue(10.0, "S1", "C1");
```

```
        dataset.addValue(4.0, "S1", "C2");
        dataset.addValue(15.0, "S1", "C3");
        dataset.addValue(14.0, "S1", "C4");
        dataset.addValue(-5.0, "S2", "C1");
        dataset.addValue(-7.0, "S2", "C2");
        dataset.addValue(14.0, "S2", "C3");
        dataset.addValue(-3.0, "S2", "C4");
        dataset.addValue(6.0, "S3", "C1");
        dataset.addValue(17.0, "S3", "C2");
        dataset.addValue(-12.0, "S3", "C3");
        dataset.addValue(7.0, "S3", "C4");
        dataset.addValue(7.0, "S4", "C1");
        dataset.addValue(15.0, "S4", "C2");
        dataset.addValue(11.0, "S4", "C3");
        dataset.addValue(0.0, "S4", "C4");
        dataset.addValue(-8.0, "S5", "C1");
        dataset.addValue(-6.0, "S5", "C2");
        dataset.addValue(10.0, "S5", "C3");
        dataset.addValue(-9.0, "S5", "C4");
        dataset.addValue(9.0, "S6", "C1");
        dataset.addValue(8.0, "S6", "C2");
        dataset.addValue(null, "S6", "C3");
        dataset.addValue(6.0, "S6", "C4");
        dataset.addValue(-10.0, "S7", "C1");
        dataset.addValue(9.0, "S7", "C2");
        dataset.addValue(7.0, "S7", "C3");
        dataset.addValue(7.0, "S7", "C4");
        dataset.addValue(11.0, "S8", "C1");
        dataset.addValue(13.0, "S8", "C2");
        dataset.addValue(9.0, "S8", "C3");
        dataset.addValue(9.0, "S8", "C4");
        dataset.addValue(-3.0, "S9", "C1");
        dataset.addValue(7.0, "S9", "C2");
        dataset.addValue(11.0, "S9", "C3");
        dataset.addValue(-10.0, "S9", "C4");
        JFreeChart chart = ChartFactory.createBarChart3D("Bar Ch
art",
            "Category", "Value", dataset, PlotOrientation.VERTIC
AL, true,
            true, false);
        return chart;
    }
    /**
    * Creates a sample time series chart.
    *
    * @return a time series chart.
    */
    private JFreeChart createTimeSeriesChart() {
        // here we just populate a series with random data...
        TimeSeries series = new TimeSeries("Random Data");
        Day current = new Day(1, SerialDate.JANUARY, 2001);
        for (int i = 0; i &lt; 100; i++) {
            series.add(current, Math.random() * 100);
```

```
            current = (Day) current.next();
        }
        XYDataset data = new TimeSeriesCollection(series);
        JFreeChart chart = ChartFactory.createTimeSeriesChart(
            "Time Series Chart", "Date", "Rate", data, true, tru
e, false);
        return chart;
    }
}
```

下一章讲述servlet的支持文件，与如何部署它们。

19.6 支持文件

Servlet为客户端产生典型的输出。大部分web应用之少包含一个HTML文件，用来进入应用的入口。本章演示的servlet，使用index.html页面，代码如下：

```
<HTML>
<HEADER>
    <TITLE>JFreeChart : Basic Servlet Demo</TITLE>
</HEADER>
<BODY>
    <H2>JFreeChart: Basic Servlet Demo</H2>
    <P>There are two sample servlets available:
    <ul>
        <li>a very basic servlet to generate a <a
        href="servlet/ServletDemo1">bar chart;</li>
        <li>another servlet that allow you to select one of <a
        href="chart.html">three sample charts. The selected char
t is
        displayed in an HTML page.</li>
    </ul>
    </BODY>
</HTML>
```

该页面上有两个链接，一个是实例1（ServletDemo1），第二个链接是两一个HTML页面，chart.html。代码如下：

```
<HTML>
<HEADER>
    <TITLE>JFreeChart Servlet Demo 2</TITLE>
</HEADER>
<BODY>
    <H2>JFreeChart Servlet Demo</H2>
    <P>Please choose a chart type:
    <FORM ACTION="servlet/ServletDemo2" METHOD=POST>
        <INPUT TYPE="radio" NAME="chart" VALUE="pie" CHECKED> Pi
e Chart
        <INPUT TYPE="radio" NAME="chart" VALUE="bar"> Bar Chart
        <INPUT TYPE="radio" NAME="chart" VALUE="time"> Time Seri
es Chart
        <INPUT TYPE="submit" VALUE="Generate Chart">
    </FORM>
</BODY>
</HTML>
```

第二个HTML页面包含<FORM>元素用来为第二个serlvet指定一个参数。当servlet运行时，返回自己的HTML，THML包含一个<IMG>元素，该元素引用了ServletDemo2ChartGenerator 的servlet。

19.7 部署Servlets

完成上面实例代码的编译厚，需要将它们连同支持文档部署到servlet引擎上，以便于客户端能够正确访问。幸运的是，这些都非常容易做到。

首先是配置web.xml文件，该文件用来描述web应用部署。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
    <servlet>
        <servlet-name>ServletDemo1</servlet-name>
        <servlet-class>demo.ServletDemo1</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>ServletDemo2</servlet-name>
        <servlet-class>demo.ServletDemo2</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>ServletDemo2ChartGenerator</servlet-name>
        <servlet-class>demo.ServletDemo2ChartGenerator</servlet-
class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletDemo1</servlet-name>
        <url-pattern>/servlet/ServletDemo1</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>ServletDemo2</servlet-name>
        <url-pattern>/servlet/ServletDemo2</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>ServletDemo2ChartGenerator</servlet-name>
        <url-pattern>/servlet/ServletDemo2ChartGenerator</url-pa
ttern>
    </servlet-mapping>
</web-app>
```

该文件通过名字列出了全部的sevlets，并且指定了具体类。实际的类被放置在servlet引擎的指定目录下面。

最后的步骤是将全部的文档拷贝到响应的servlet引擎的目录下面。我们使用的是servlet引擎是Tomcat。在Tomcat下的webapps目录下面创建一个jfreechart2的目录，将index.html和chart.html文件拷贝到下面的目录：

```
webapps/jfreechart2/index.html
webapps/jfreechart2/chart.html
```

接下来，在目录jfreechart2下建立一个子目录WEB-INFO，将web.xml文件拷贝到该目录下面。

```
webapps/jfreechart2/WEB-INF/web.xml
```

WEB-INFO目录下面创建子目录classes/demo将编译的类放在该目录下面。

```
webapps/jfreechart2/WEB-INF/classes/demo/ServletDemo1.class
webapps/jfreechart2/WEB-INF/classes/demo/ServletDemo2.class
webapps/jfreechart2/WEB-INF/classes/demo/ServletDemo2ChartGenera
tor.class
```

最后，将相关的jar拷贝到下面目录：

```
webapps/jfreechart2/WEB-INF/lib/jcommon-1.0.9.jar
webapps/jfreechart2/WEB-INF/lib/jfreechart-1.0.6.jar
```

现在启动我们的servlet引擎，在我们的浏览器中输入：

http://localhost:8080/jfreechart2/index.html

如果全部文件放置在适当位置，而不出现特殊意外的话，我们将会看到上面图19.2所示的界面。

20 JFreeChart相关技术

20.1 简介

本章节主要介绍了JFreeChart涉及的各种信息。

20.2 X11/Headless Java

如果我们在Unix/Linux上的服务环境使用JFreeChart，我们会遇到JFreeChart在没有X11的情况下不能运行。这与运行在AWT上的java代码是同一个问题。更多的信息见下面链接：

http://java.sun.com/products/java-media/2D/forDevelopers/java2dfaq.html#xvfb

同时在JFreeChart的论坛里面也有好多的信息，可以找到一些额外的思路：

http://www.jfree.org/phpBB2/viewtopic.php?t=1012

20.3 JSP

如果开发者在JSP中使用JFreeChart比较感兴趣，那么可以从下面网址找到更多信息：

http://cewolf.sourceforge.net/

20.4 加载图片

图像在Java中是用Image来描述的。我们可以使用开发包里面的createImage()方法来创建图像，但是我们需要意识到该方法加载图像时不是同步的——换句话说，方法返回的与图像加载是在不同的线程中。这样会产生这样一个问题就是在图像为完全加载完成时，我们就使用了该图像。

我们可以使用MediaTracker类来检查图像加载的进度。但万一在某个地方我们需要在使用方法之前，必须确保图片完全加载完毕。这时该怎么办呢？使用Swing的ImageIcon类可以解决我们这个问题，代码如下：

```
ImageIcon icon = new ImageIcon("/home/dgilbert/temp/daylight.png
");
Image image = icon.getImage();
```

构造方法直到图片完全加载完成后才返回，因此在我们调用方法getImage()时，下图已经图像加载完毕。

21 包

21.1 概述

下面内容讲述了JFreeChart类的参考信息。

包名	说明
org.jfree.chart	主图表类
org.jfree.chart.annotations	注释图表的简单框架
org.jfree.chart.axis	轴类和相关接口
org.jfree.chart.editor	为图表提供的属性编辑器框架（不完善）
org.jfree.chart.encoders	写图象文件类
org.jfree.chart.entity	描述图表实体的类
org.jfree.chart.event	事件类
org.jfree.chart.imagemap	HTML图片映像工具类
org.jfree.chart.labels	图表标签和信息提示类
org.jfree.chart.needle	Needle classes for the compass plot
org.jfree.chart.plot	Plot类和接口
org.jfree.chart.renderer	Renderer的基本类包
org.jfree.chart.renderer.category	Plug-in renderers for use with the CategoryPlot class.
org.jfree.chart.renderer.xy	Plug-in renderers for use with the XYPlot class.
org.jfree.chart.servlet	Servlet utility classes.
org.jfree.chart.title	图表标题类
org.jfree.chart.urls	在图像映像区产生URLs的接口和类
org.jfree.chart.util	实用工具类
org.jfree.data	Dataset接口和类
org.jfree.data.category	CategoryDataset接口和相关类
org.jfree.data.contour	ContourDataset接口和相关类
org.jfree.data.function	Function2D接口和相关类
org.jfree.data.gantt	甘特图的dataset接口和类
org.jfree.data.general	通用的dataset类

org.jfree.data.io	通用的dataset的I/O类
org.jfree.data.jdbc	JDBC相关的dataset类
org.jfree.data.statistics	产生统计的相关类
org.jfree.data.time	基于时间的dataset接口和类
org.jfree.data.time.ohlc	展示高低开发图表dataset的类
org.jfree.data.xml	从xml文件读取dataset的类
org.jfree.data.xy	XYDataset接口和相关类

更多的信息可以查看javadoc产生的HTML文档。